

MSc THESIS

Software solution for Entropy Decoding on TM32 cores

Arjen Westerterp

Abstract



CE-MS-2003-13

Entropy Decoding is an essentially sequential task. Executing this task on a processor that benefits from Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) or both requires an efficient implementation of Entropy Decoding. Entropy Decoding forms the part of MPEG-2 Decoding that exploits the least parallelism. Creating more parallelism in Entropy Decoding is expected to optimize MPEG-2 Decoding significantly. When writing an efficient MPEG-2 Decoder the result needs to be conform to the Berkeley MPEG-2 Decoder. In this thesis the Entropy Decoding process is optimized by compressing the data-stream from the Lookup Table to the Variable Length Decoder. In addition, the number of branches in the Entropy Decoding process is reduced. This stimulates Instruction Level Parallelism (ILP), that is exploited even more by using pre-loading data. The result of these three optimization steps is evaluated on the TriMedia32 core. Here a reduction of 30% in the number of cycles needed to decode a coefficient is achieved when going from 23 to 16 instruction cycles. In addition, an exceptionally high VLIW slot occupancy of more than 85% shows that the Entropy Decoding process is highly optimized. This result is integrated in the MPEG-2 Decoder written for SpaceCAKE. The resulting MPEG-2 Decoder

has the same properties in terms of DLP compared to the original decoder. Despite the patch needed to conform to the data management in the multi-threaded MPEG-2 decoder, the improvement still is significant. On average an improvement of 12% is measured, with a peak improvement of 20%. These figures represent the lower bound of performance improvement. It is required to rewrite the data management structure of the Berkeley MPEG-2 Decoder to allow for a fully performing optimized Entropy Decoding process in this decoder. In conclusion, the optimized MPEG-2 Decoder gives a significant performance improvement based on remarkable results achieved by increasing the efficiency of the compiled Entropy Decoder source code, thus optimizing the Entropy Decoder process.

Software solution for Entropy Decoding on TM32 cores

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Arjen Westerterp
born in Stirling, United Kingdom

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Software solution for Entropy Decoding on TM32 cores

by Arjen Westerterp

Abstract

Entropy Decoding is an essentially sequential task. Executing this task on a processor that benefits from Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) or both requires an efficient implementation of Entropy Decoding. Entropy Decoding forms the part of MPEG-2 Decoding that exploits the least parallelism. Creating more parallelism in Entropy Decoding is expected to optimize MPEG-2 Decoding significantly. When writing an efficient MPEG-2 Decoder the result needs to be conform to the Berkeley MPEG-2 Decoder. In this thesis the Entropy Decoding process is optimized by compressing the data-stream from the Lookup Table to the Variable Length Decoder. In addition, the number of branches in the Entropy Decoding process is reduced. This stimulates Instruction Level Parallelism (ILP), that is exploited even more by using pre-loading data. The result of these three optimization steps is evaluated on the TriMedia32 core. Here a reduction of 30% in the number of cycles needed to decode a coefficient is achieved when going from 23 to 16 instruction cycles. In addition, an exceptionally high VLIW slot occupancy of more than 85% shows that the Entropy Decoding process is highly optimized. This result is integrated in the MPEG-2 Decoder written for SpaceCAKE. The resulting MPEG-2 Decoder has the same properties in terms of DLP compared to the original decoder. Despite the patch needed to conform to the data management in the multi-threaded MPEG-2 decoder, the improvement still is significant. On average an improvement of 12% is measured, with a peak improvement of 20%. These figures represent the lower bound of performance improvement. It is required to rewrite the data management structure of the Berkeley MPEG-2 Decoder to allow for a fully performing optimized Entropy Decoding process in this decoder. In conclusion, the optimized MPEG-2 Decoder gives a significant performance improvement based on remarkable results achieved by increasing the efficiency of the compiled Entropy Decoder source code, thus optimizing the Entropy Decoder process.

Laboratory : Computer Engineering
Codenummer : CE-MS-2003-13

Committee Members :

Advisor and Chairperson: Stamatis Vassiliadis, CE, TU Delft
Co-advisor: Jos T.J. van Eijndhoven, IT, Philips Research
Member: Paul Stravers, IT, Philips Research
Member: Mihai Sima, CE, TU Delft
Member: Ben H.H. Juurlink, CE, TU Delft

To wisdom in all explorable fields

Contents

List of Figures	vii
List of Tables	ix
Acknowledgements	xi
1 Introduction	1
1.1 Current status	1
1.2 Motivation	3
1.3 Challenges	4
1.4 Overview	5
2 Theoretical background	7
2.1 Run Length Code	7
2.2 Variable-Length Code	8
2.3 Entropy Decoding	13
2.4 MPEG-2 organization	15
2.5 Theoretical Background Conclusion	17
3 TriMedia & SpaceCAKE	19
3.1 TriMedia Architecture and Tool-set	19
3.2 SpaceCAKE	24
3.3 TriMedia & SpaceCAKE Conclusion	25
4 Stand-alone Entropy Decoding on TM32	27
4.1 Porting TM64 specific code for Entropy Decoding	27
4.2 Re-ordering data organization in the Lookup Table	32
4.3 Entropy Decoding Organization	36
4.4 Identification of unique cases	36
4.5 Removal of data update branches	39
4.6 Pre-loading data	40
4.7 Stand-alone Entropy Decoding on TM32 Conclusion	41
5 Entropy Decoder Integration in Multi-threaded MPEG-2 Decoder	43
5.1 Identification of Entropy Decoder blocks	43
5.2 Adapting data management	44
5.3 Inserting initializations and Lookup Tables	45
5.4 Merging Entropy Decoding into two functions	46
5.5 Obstacles encountered during development phase	46
5.6 Entropy Decoder Integration Conclusion	47

6	Experimental results	49
6.1	Experimental framework	49
6.2	Stand-alone Entropy Decoder performance	49
6.3	MPEG-2 decoder on SpaceCAKE	58
6.4	Experimental results Conclusion	64
7	Conclusion	65
	Bibliography	67

List of Figures

1.1	TriMedia/CPU32 overview of core and its peripherals	3
1.2	A SpaceCAKE instance of eight TM32 cores and one MIPS	4
2.1	Variable-Length decoding principle [14]	14
2.2	A schematic overview of the MPEG-2 decoding process	15
2.3	Hierarchical organization of the MPEG-2 video bit-stream	16
3.1	TM1000 block diagram An example of the TriMedia	20
3.2	TM1000 DSPCPU-organization	22
3.3	A Typical architecture of a SpaceCAKE instance	24
4.1	The flowchart of the variable-length decoding procedure	30
4.2	Filling of crucial bit positions	34
4.3	The complete filling of bit positions	35
4.4	Entropy Decoding organization scheme	36
6.1	Entropy Decoding in cycles per coefficient with 64-bit data from the Lookup Table	52
6.2	Entropy Decoding in issues per cycle with 64-bit data from the Lookup Table	53
6.3	Entropy Decoding in cycles per coefficient with 32-bit data from the Lookup Table	54
6.4	Entropy Decoding in issues per cycle with 32-bit data from the Lookup Table	55
6.5	Entropy Decoding in cycles per coefficient with pre-loading of data from the Lookup Table	56
6.6	Entropy Decoding in issues per cycle with pre-loading of data from the Lookup Table	57
6.7	Number of cycles in Entropy Decoding and its accompanying functions for queen	59
6.8	Number of cycles in Entropy Decoding and its accompanying functions for Popplen	60
6.9	Average number of cycles needed for MPEG-2 decoding depending on the number of processors	61
6.10	Speedup of MPEG-2 decoding per added processor	62
6.11	Speedup of optimized MPEG-2 decoding per added processor	63

List of Tables

1.1	Trade off between hardware and software solution	2
2.1	Run Length Coding on a four by four matrix	8
2.2	Optimum Binary Coding Process	11
2.3	A table organized Huffman tree	11
2.4	A table organized Single side Growing Huffman tree	12
2.5	Probabilities and Huffman code for the dataset from Table 2-2	12
2.6	Lookup Table definitions for Variable-Length Decoding	13
3.1	Mapping from optimization level to optimizations [2]	23
4.1	Number of address lines, size and offset for each Lookup Table	29
4.2	Overview of data compression effects on the number of cycles needed for involved operations	33
4.3	Overview of occurrence of the VD, EOB and ESC flags	35
6.1	Overview on coefficient density per block for input bit-streams	58

Acknowledgements

The research described in this thesis is produced with the support of many people without whom it would have been much harder to produce this dissertation.

First, I want to thank my family for their understanding and support during my studies in Delft. Their non-technical background and knowledge of university education have broadened my view on my subject.

I want to thank Prof. Dr. Stamatis Vassiliadis for giving me the opportunity to do my masters research outside of the Technical University of Delft. This gave me the chance to broaden my view in the Computer Engineering research field.

I want to thank Dr. Ir. Jos T.J. van Eijndhoven for giving me the opportunity to do research at the Philips Research Laboratories. The research on the TriMedia processor offered me the chance to build software with a realistic view. This also broadened my insight in commercially oriented research.

Especially I want to thank Mihai Sima for all his effort and time spent on assisting me throughout my research. I am very pleased with all the comments, ideas and thoughts during the development of my masters project. Even in the last days of his own research, he found time to have a thorough look on my writing and presentation of results.

I want to thank Dr. Ir. Paul Stravers and Dr. Ir. Jan Hoogerbrugge for making SpaceCAKE hard- and software available for my research. This has given me more insight in the steps I took.

Finally, I want to thank my friends, officemates and the members of the Coffee Club whose company I enjoyed a lot. The regular daily support of fresh coffee, laughs and discussions have found their way into this thesis.

Arjen Westerterp
Delft, The Netherlands
September 28, 2003

Introduction

T*his chapter shows how and where the research done for this thesis can be placed. It explains the origin of multimedia and how this has evolved in the past. Results achieved in research from the past are given for comparison reasons and to give a background on this field of research. The motivation for the research is based on the different properties for a soft- and hardware solution. Challenges in this research are coming from the VLIW principle and porting source code.*

The chapter starts with the current status in the research field (section 1.1). Section 1.2 motivates the research done for this thesis. The challenges in the research are explained in section 1.3. The last section (1.4) gives an overview on the whole thesis.

1.1 Current status

The starting point of multimedia was at the time when audio and video became available in digital format. Multimedia gave opportunities to use computer-like facilities in a digital form not only for audio and video but also for text, figures and tables. These media all can be treated and stored in the same digital manner. Storing or transmitting digital audio requires large amounts of information bandwidth. For example, National Television Systems Committee resolution MPEG-2 decoding takes more than 400 MOPS and the encoding takes 30 GOPS. These results are achieved with a non-optimized MPEG-2 tool-set. A reduction is achieved when using compression techniques. For example a completely filled regular audio CD fits ten times on the same disc when the MPEG-1-layer3 (mp3) compression format is used. The MPEG-1 compression standard comes from the Moving Pictures Expert Group and is further developed to result in the MPEG-2 compression standard. This thesis focuses on MPEG-2 decompression techniques.

Entropy Decoding of a bit-stream of pictures along with its control data is the most sequential part of MPEG-2 Decoding. Moreover, Entropy Decoding lies on the critical path of the MPEG-2 Decoding process. The Entropy Decoder consists of Variable-Length Decoder that is followed by Run Length Decoder. Both perform an essentially sequential task in which only a little parallelism is present. Therefore, Entropy Decoding is an intricate function on processors that benefit from parallelism. A multimedia processor that benefits from parallelism is the TriMedia/CPU32. This processor is targeted for real-time processing of multimedia streams. A TriMedia/CPU32 is based on the Very Large Instruction Word principal. This allows Multiple Instructions Multiple Data MIMD operations. A Pentium with MMX (multimedia extension) uses Single Instruction Multiple Data SIMD operations only [3].

An Entropy Decoder for a VLIW processor is presented in [14]. This Entropy Decoder written in software and targeted for the TriMedia/CPU64. The TriMedia/CPU64 cur-

rently only exists in software. It has a machine-level simulator, compiler and scheduler that execute and analyze source code written for this processor. The TriMedia/CPU64 is an extended version of the TriMedia/CPU32 [14]. The extensions are the doubled register size, data-cache size and instruction-cache size. In addition super-operations are added. These operations are executed on two merged VLIW issue slots. This gives the opportunity to use twice the number of input and output registers for an instruction.

An implementation of MPEG-2 compliant Entropy Decoding in software is comparable to results from other platforms. The results of these implementations give a framework in which the outcome of this thesis can be placed. Possible platforms to compare these results with are:

- TriMedia/CPU32
- TriMedia/CPU64
- The TMS320C80 media video processor by Texas Instruments
- A Pentium processor with MMX

The latter two consider Inverse Quantization (IQ) as well. The additional IQ functionality is not a real concern in the TriMedia/CPU32 case. Results have shown that a significant number of operations related to IQ can still be scheduled in the delay slots of the table lookup [14]. The results of Decoding one Discrete Cosine Transform coefficient on these platforms are:

- 23,0 cycles per coefficient for TriMedia/CPU32 [from section 6.2]
- 16,9 cycles per coefficient for TriMedia/CPU64 [14]
- 26,0 cycles per coefficient for TMS320C80 [6]
- 33,0 cycles per coefficient for Pentium MMX [3]

The term cycles per coefficient should be read as the number of cycles needed to decode one Discrete Cosine Transform coefficient.

Table 1.1: Trade off between hardware and software solution

Hardware Solution	Software Solution
<ul style="list-style-type: none"> • Performing because of dedicated circuits • Not flexible because of fixed structures 	<ul style="list-style-type: none"> • Less performing on general purpose hardware • Flexible because of reprogram ability

1.2 Motivation

Entropy Decoding, specifically Variable Length Decoding has been implemented in full-custom hardware in the current TriMedia/CPU32. Figure 1-1 shows the full-custom hardware available to the TriMedia32 core[15]. This hardware is located in the coprocessors along the Main Memory Interface.

Full-custom hardware implemented in coprocessors is built to execute specific tasks that are time consuming. This relieves the TriMedia32 core of tedious and time-consuming tasks. The disadvantage to this solution is the lack of flexibility. A hardwired solution needs to be redesigned for each new task it is expected to execute. This may result in a problem with rapidly evolving standards like in the multimedia domain. This gives a trade off between the hardware and the software solution, which is explained in Table 1-1.

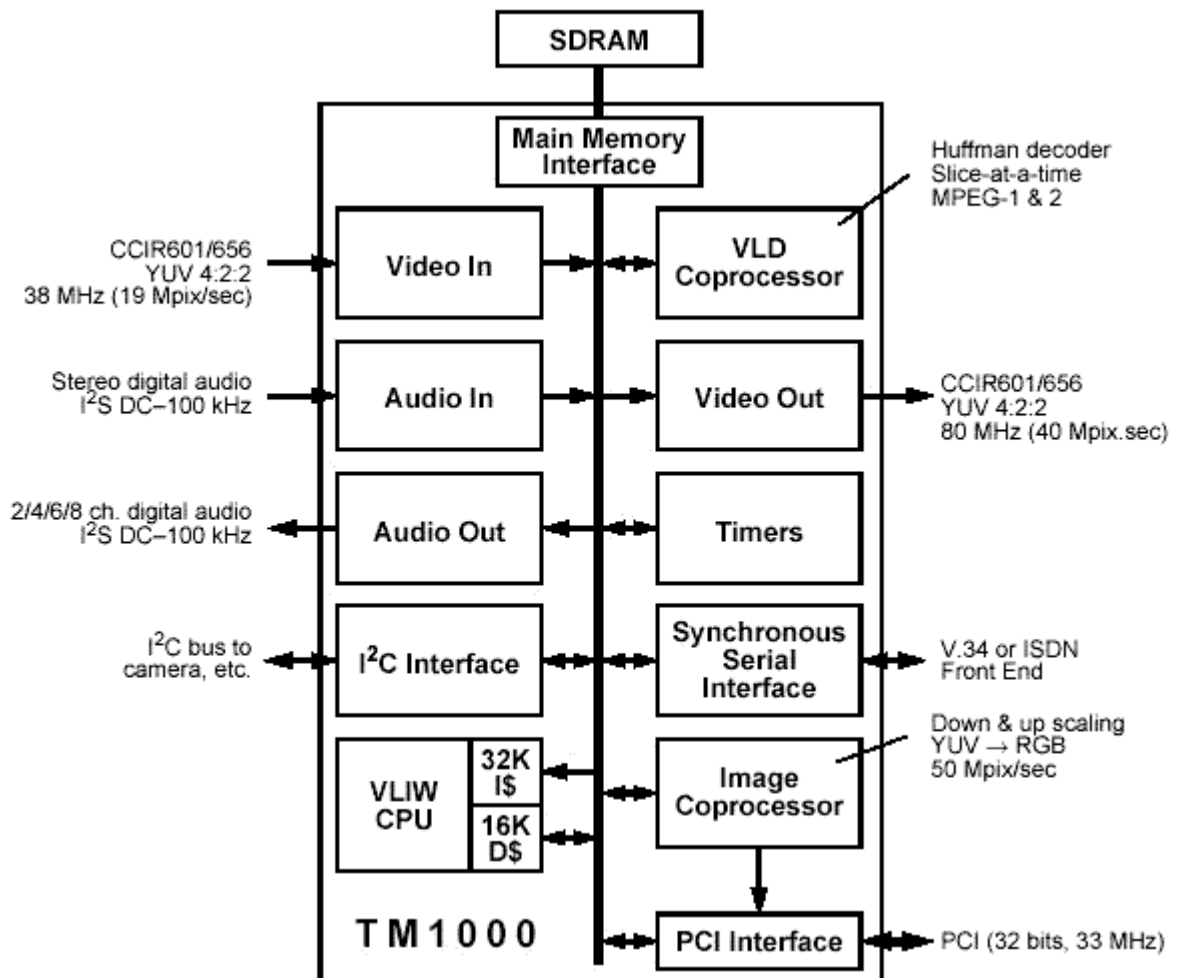


Figure 1.1: TriMedia/CPU32 overview of core and its peripherals

Coprocessors like the VLD coprocessor support the TriMedia32 core. Another way to support a TriMedia32 core is surrounding it with more TriMedia32 cores. This creates an architecture that is based on the principles of a SpaceCAKE architecture [12]. In such architecture, a number TriMedia32 cores and MIPS's are connected through an interconnection network. A possible instance of this architecture is shown in figure 1-2. Here eight TriMedia32 cores and one MIPS are connected through an interconnection network. The resulting architecture is more flexible because of its programmability. This flexibility meets the demands like in the multimedia domain. In this domain, MPEG-2 decoding is used to visualize compressed audio and video. MPEG-2 decoding consists of several steps that are explained in section 2.4.

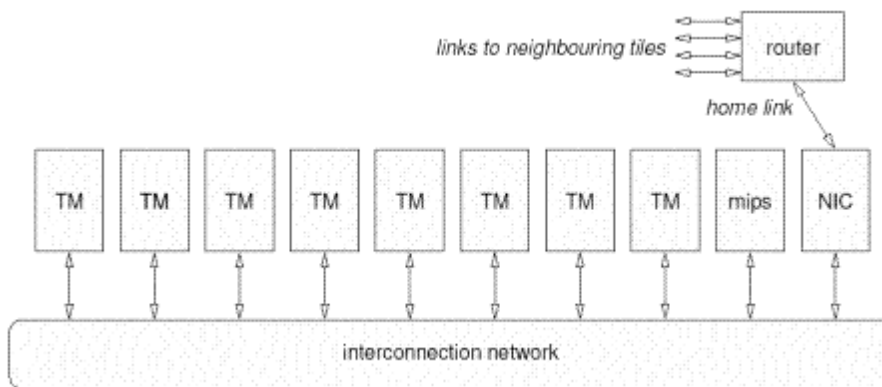


Figure 1.2: A SpaceCAKE instance of eight TM32 cores and one MIPS

One of these steps is Entropy Decoding. There are three reasons to optimize Entropy Decoding: Entropy Decoding is

- Time consuming
- Essentially sequential
- On the critical path of MPEG-2 decoding

These reasons show that it is obvious that an efficient implementation of Entropy Decoding on the TriMedia32 core is needed. This is the subject of this thesis.

1.3 Challenges

A regularly used MPEG-2 decoder is the Berkeley MPEG-2 decoder. This decoder is a conformance tool against which a new implementation of MPEG-2 decoding needs to be checked. This makes the Berkeley MPEG-2 decoder a good starting point for optimization steps also when only a part of the MPEG-2 decoder is optimized. In such way, writing a complete MPEG-2 decoder from scratch can be avoided. In the previous

section the need of an efficient implementation of Entropy Decoding for the TriMedia32 core is shown. A comparable approach was taken when the Entropy Decoder for the TriMedia64 core [11] was written. The result of this approach shows options of how to optimize an Entropy Decoder for the TriMedia32 core. Four key issues need to be tackled to build this optimized Entropy Decoder:

1. Parallelism is not evidently present in Entropy Decoding since it is an essentially sequential task. The TriMedia32 core benefits from Instruction Level Parallelism (ILP) like all VLIW platforms do. Therefore, all forms of parallelism present in Entropy Decoding need to be revealed. This is expected to show an improvement in VLIW slot occupancy. Creating extra forms of parallelism can extend this improvement.
2. The TriMedia64 core has the possibility to use super operations that are not available to the TriMedia32 core. These operations are based on two merged VLIW issue slots resulting in a doubled capacity of input and output registers. A reduction in data bandwidth and a reduction in available input and output registers needs to be tackled in new performing operations.
3. The Lookup Table available from the TriMedia64 Entropy Decoder is based on a character organization. The characters are packed into eight-byte vectors. This generates the possibility of using vector operations specific to the TriMedia64 core. Although vector operations are present in the TriMedia32 core, these do not process eight-byte vectors. Therefore, the eight-byte vectors need to be analysed for compressibility.
4. When VLIW issue slots appear to remain unused, these can be filled with newly created instructions. These need to be independent of the current running process at that point. The new instructions should result in earlier execution of tasks that support current or expected operations. In such way, extra ILP is created and possible wait cycles are removed.

1.4 Overview

Chapter 2 gives a theoretical background on the methods that need to be optimized. As stated in section 1.2, an efficient implementation of Entropy Decoding for the TriMedia32 core is needed. Entropy Decoding consists of two parts. These parts are Variable Length Decoding followed by Run Length Decoding. Each decoding step is explained by its counter part. This shows how a Variable Length Code is built and the theory behind it. Section 2.4 shows the place of VLD, RLD and Entropy Decoding in the MPEG-2 decoding procedure.

Chapter 3 explains the architecture of the TriMedia and the SpaceCAKE tool-set. The TriMedia/CPU32 is a multimedia processor that consists of a TriMedia32 core and a set of coprocessors, to relief the core from time-consuming tasks (see section 1.2). Another way to achieve this goal is by adding more TriMedia32 cores to build a network of TriMedia32 cores. This is done in SpaceCAKE. A possible instance of SpaceCAKE

is eight TriMedia32 cores and one MIPS connected through a network that builds a performing tile for random multimedia applications.

Chapter 4 shows the steps taken to optimize a stand-alone Entropy Decoder on TriMedia32. These steps are based on the challenges presented in section 1.3. At first, the TriMedia64 source code is rewritten for the TriMedia32 core. From this a set of three possible shifting methods emerges; a choice from this is made in a latter phase. Tackling the third challenge compresses the dataflow from the Lookup Table from 64 bits with redundancy to 32 bits without redundancy. The last step handles the introduction of more parallelism by pre-loading data into registers. Chapter 5 shows the steps taken to integrate the optimized Entropy Decoder written for TriMedia32 in a Multi-threaded Berkeley MPEG-2 decoder. Integration requires identification of parts of source code that need to be replaced by the optimized source code. After replacement, adaptations in the source code are made to evolve stand-alone data management and initializations to an integrated entity. The resulting entities are merged into two functions that cover the whole Entropy Decoding process. Within the Entropy Decoding process care is taken care to create pictures for a moving picture as the final result.

Chapter 6 shows the experimental results per phase throughout the optimization and integration of Entropy Decoding. Reducing redundancy in the data that comes from the Lookup Table and pre-loading data show how to optimize Entropy Decoding. This result is integrated into the SpaceCAKE MPEG-2 Decoder where it gives a significant optimization.

In Chapter 7 it is concluded that the optimized Entropy Decoder is highly performing on the TriMedia32 core. The integration of the Entropy Decoder in the SpaceCAKE MPEG-2 Decoder shows a significant improvement in performance. Even despite of a patch needed for integration in an MPEG-2 Decoder. This implies that rewriting the data management structure of the SpaceCAKE MPEG-2 Decoder is required. This allows for a fully performing optimized Entropy Decoder in the MPEG-2 Decoder.

Theoretical background

This chapter gives an introduction in methods and processors used throughout this thesis. The methods being parts of MPEG-2 decoding are:

- *Run Length Decoding*
- *Variable-Length Decoding*
- *Entropy Decoding*

The processors are:

- *TriMedia*
- *SpaceCAKE*

The three Decoding methods are used in both Philips processors TriMedia and SpaceCAKE. When introducing TriMedia, the Very Long Instruction Word is highlighted. This forms the basis of all optimizations discussed in this thesis.

The chapter introduces the Run Length code in section 2.1. From this Run Length code a Variable Length code is made as explained in section 2.2. These two compression techniques form a part of Entropy Decoding as shown in section 2.3. Since Entropy Decoding is a part of MPEG-2 Decoding, this is highlighted in section 2.4.

2.1 Run Length Code

Run Length Coding (RLC) is a compression technique that like others decreases redundancy in data representation. The goals of minimizing redundancy are:

- Decrease data storage requirements
- Decrease data communication costs

RLC operates on a vector. This vector is the result of a zigzag transformation of a square matrix. Assume that in zigzag transformation the next vector element is given by:

1. The first element on the left top of the matrix
2. The element placed right of the first element
3. The element either diagonally downwards to the left or diagonally upwards to the right. (In all cases only one of these moves is possible)

4. The element right of the current one at the first or last row; and continue with step 3
5. The element below the current one at the first or last column; and continue with step 2

With these rules it is possible to code a square matrix with Run Length Coding as well. The coding procedure specifies the number of zeros (run) that precede a value (level) in the vector. In this way, run-level pairs are made. When all remaining elements in a vector are zero, an End Of Block code is generated. Figure 2-1 shows an example of Run Length Coding performed on a matrix.

Table 2.1: Run Length Coding on a four by four matrix

<i>Matrix</i>	<i>Vector</i>	<i>Run</i>	<i>Level</i>
2 0 13 0	2	0	2
0 0 0 0	0		
0 5 0 7	0		
8 0 0 0	0		
	0		
	13	4	13
	0		
	0		
	5	2	5
	8	0	8
	0		
	0		
	0		
	7	3	7
	0	<i>EOB</i>	
	0		

2.2 Variable-Length Code

Variable-Length Code, also called Huffman Code is another way to reduce redundancy. This requires a finite number of messages. This Huffman Coding method has the following properties [8]:

1. If there are more messages than symbols available, some messages will consist of multiple symbols.
2. There are no two messages consisting of an identical arrangement of symbols.

3. Message codes are constructed in such a way that no additional indication is needed to specify where a message code begins or ends.

The second and third property imply that no message should be coded in such a way that any message code is the first part of any message code of longer length. Probably the most familiar example of a Huffman Code is in the phrase: "One if by land and two if by sea." In this case, the messages are "by land" and "by sea." The message codes are "one" and "two." This set of message codes can be put in any way in a sequence of symbols without having an effect on the transmitted information. E.g., '112221211221122' is interpretable in one way only. The set of messages can be extended with another ten messages for example. A way of choosing the message codes is taking the digits zero to eleven. A transmitted sequence of symbols can be 0123456789110. The last three digits can be seen in three ways:

- 1 1 0
- 11 0
- 1 10

This violates the third property of Huffman coding. Choosing double digits for the first two symbols solves the indicated problem. When having built this Huffman Code, the next step is to make it optimal. For an optimum Huffman Code, the length of a given message code cannot be smaller than the length of a more probable message code. If this requirement is not met, the conflicting message codes can be interchanged to assure that the most probable message gets the shortest message code. When there are several messages with an identical probability, the length of their message codes may differ. Interchanging these message codes will not affect the average code length of the messages transmitted. In this way, messages are ordered by probability. This gives the following additional Huffman Coding properties [8]:

4. The length of any message code is smaller or equal than the length of a less probable message code.
5. The length of the two least probable message codes is identical.
6. There exist at least two, and not more than all-possible, message codes that are alike except for their final digits.
7. Each possible sequence of symbols either must be a message code or must have one of its prefixes used as a message code.

With these seven properties, a Huffman tree can be built. As an example, a binary Huffman tree is discussed below. A Huffman tree consists of [9]:

- One root
 - The start of the Huffman tree where no symbol has been de- or encoded yet
- Children
 - This node has children and/or leaves connected to it.
 - The probability of a child is the sum of the probabilities of its children and leaves.
- Leaves
 - This is an end in the tree. At this point, the relation between message code and message is found.
- Level
 - This indicates a group of children and/or leaves with (almost) equal probability
 - Property 5 shows there is a combination of at least two children and/or leaves in a level.
- Nodes
 - This is the group name of Root, Children, and Leaves.

When a set of data and its probability is given, a Huffman tree can be constructed. Concern a dataset as given in Table 2-2.

The one but left column in Table 2-2 shows the original probabilities of the data set. This is called the '*original data ensemble*'. All the columns right of the '*original data ensemble*' show an '*auxiliary data ensemble*'. In an '*auxiliary data ensemble*' the two nodes with the smallest probability are replaced by a new *node* or in the last case the *root*. This new *node* is shown in a grey box. This new node is placed in the same order of probability among the other present nodes. When finally the *root* with probability 1.00 has been made, the tree is complete. Following the path from root to the leaves builds a Huffman tree. The result is schematically shown in Table 2-3. This result almost shows a single side growing Huffman tree. Re-ordering data gives the result as shown in Table 2-4. This process does not affect the average code length in a sequence of messages.

Table 2.2: Optimum Binary Coding Process

Data Probabilities													
Node	Original Data Ensemble	Auxiliary Data Ensembles											
		1	2	3	4	5	6	7	8	9	10	11	12
A													1.00
B												0.60	
C											0.40	0.40	
D										0.36	0.36		
E									0.24	0.24	0.24		
F								0.20	0.20	0.20			
G	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20			
H							0.18	0.18	0.18				
I	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18	0.18				
J						0.14	0.14	0.14					
K				0.10	0.10	0.10	0.10						
L	0.10	0.10	0.10	0.10	0.10	0.10	0.10						
M	0.10	0.10	0.10	0.10	0.10	0.10	0.10						
N	0.10	0.10	0.10	0.10	0.10	0.10							
O			0.08	0.08	0.08								
P		0.08	0.08	0.08									
Q	0.06	0.06	0.06	0.06	0.06								
R	0.06	0.06	0.06	0.06									
S		0.04	0.04	0.04									
T	0.04	0.04	0.04										
U	0.04	0.04	0.04										
V	0.04	0.04											
W	0.04	0.04											
X	0.03												
Y	0.01												

Table 2.3: A table organized Huffman tree

A															
B							C								
D				E				F			G				
H		I		J		K		L		M					
N	O			P	Q	R	S								
	T	U			V	W		X	Y						

In the Huffman tree from Table 2-2, A is the *root*, N, T, U, I, V, W, Q, R, X, Y, L, M, G are the *leaves*, and all the other nodes are *children*. The following Table 2-5 shows the average code length for this tree achieved by Huffman coding instead of straightforward encoding by binary counting. Binary counting takes an average of 5.00

bits per message. The used Huffman coding method gives an average code length of 3.42 bits per message as shown in the last row of Table 2-5.

Table 2.4: A table organized Single side Growing Huffman tree

A									
B					C				
D			E			F		G	
H		J	K		I	L	M		
O	P	S	N	Q					
T	U	V	W	X	Y				

From Table 2-4 a Huffman code is created. This needs definition of what to choose as a binary '1' and what to choose as a binary '0'. Here a branch to the left as in A to B is '0' and a branch to the right as in A to C is '1'. This results in the right column of Table 2-5.

Table 2.5: Probabilities and Huffman code for the dataset from Table 2-2

Leave	Probablility	Code length	Code
G	0.20	2	11
I	0.18	3	011
L	0.10	3	100
M	0.10	3	101
N	0.10	4	0011
Q	0.06	4	0100
R	0.06	4	0101
T	0.04	5	00000
U	0.04	5	00001
V	0.04	5	00010
W	0.04	5	00011
X	0.03	5	00100
Y	0.01	5	00101

Average code Length 3.42

When VLC and RLC are combined they form Entropy Coding for i.e. MPEG-2 coding. In Entropy Coding, RLC needs to take place before VLC. Only in this way RLC generates the run-level pairs that are Huffman coded by VLC. VLC generates from frequently occurring run-level pairs a Huffman code according to the B12, B13, B14 and B15 MPEG-2 coding tables. Rarely occurring run-level pairs do not have a Huffman code. These are coded through putting an Escape code in front of a fixed length representation of the run-level pair. In this way an Entropy Coded bit-stream emerges from the combined RLC and VLC process.

2.3 Entropy Decoding

The inverse operations of Run Length Coding and Variable-Length Coding come together in Entropy Decoding. Entropy Decoding consists of Variable-Length Decoding followed by Run Length Decoding. Both are mainly sequential tasks where RLD relies on the results of VLD. I.e. RLD and VLD run independently as long as RLD is preceded by VLD. The input of the Variable-Length Decoder is the incoming bit-stream. The output is a run-level pair and its code length. The Variable-Length Decoder generates this output according to the standardized Look-Up Tables, these are:

- B12 (512 entries)
- B13 (1024 entries)
- B14 (768 entries)
- B15 (528 entries)

Which Lookup Table is used is pre-specified by the type of image and a bit-field in the incoming bit-stream called `intra_vlc_format`. This bit-field forms a part of the macro-block header; its definition is shown in Table 2-6. The possible types of images are:

- Intra (I)
- Non-intra (NI)
- Luminance (Y)
- Chrominance (C)

Table 2.6: Lookup Table definitions for Variable-Length Decoding

Intra_vlc_format			0	1
Intra	DC coefficient	Luminance	B12	B12
		Chrominance	B13	B13
	AC coefficient		B14	B15
Non-Intra	1st & subsequent coefficients		B14	B14

The Look-Up Table receives the Variable-Length Code from the bit-stream as an address. At this address the set of output data (run-level, code length and special codes) is found (Figure 2-1).

The output data from the lookup table contains information about:

- Level: This is a non-zero coefficient at a given position in a matrix. The matrix forms part of the final picture.
- Run: The number of positions filled with zeros between two non-zero coefficients.

- Valid Decode: This bit is set to one when a valid run level pair is found in the lookup table.
- Escape: This bit is set when a number of following bits show run and level directly.
- Code: Length The number of decoded bits.
- Lookup Address Width: The number of bits that will give the next address in the lookup table.
- Table offset: The number of bits that need to be added to the lookup address width to find the next address in the lookup table.
- EOB (End Of Block): This bit is set when the last non zero coefficient in a matrix is decoded.

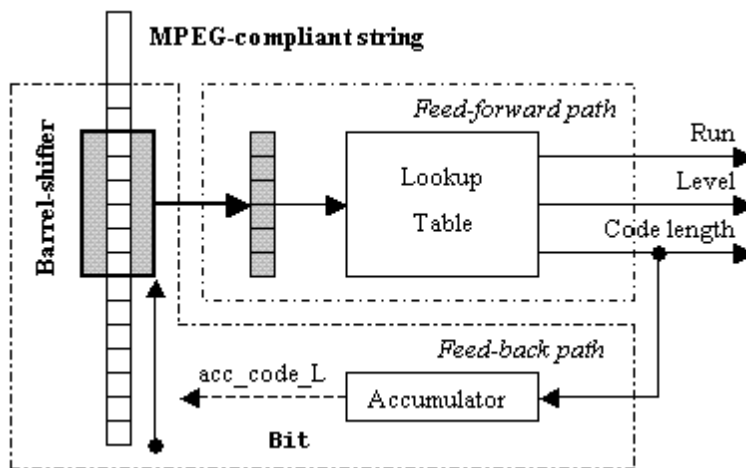


Figure 2.1: Variable-Length decoding principle [14]

The output, code length, is used in two ways:

- Determine the starting point of the next codeword in the incoming bit-stream.
- Determine what bits can be shifted out of the buffer from which the Variable-Length Code is read.

The output, run-level pair or End Of Block, is used to fill in a matrix according to the inverse of the Variable-Length coding technique as explained in section 2.2.

2.4 MPEG-2 organization

In MPEG-2 Entropy Decoding, the maximum code length is 16 bits plus a sign bit. This excludes an Escape code that contains 24 bits. Therefore, a maximum of 17 bits is coded in the Lookup Table. This Lookup Table contains ($2^{17} =$) 128 K words for a direct mapping of all possible code words. The organization of Lookup Tables in Entropy Decoding is explained in section 2.3. In general, a bit-stream uses all Lookup Tables. Switching between tables is possible per macro-block and/or block. Exceptional cases occur when:

- An Escape code is encountered; Six unsigned bits run and twelve signed bits level follow the six unsigned bits Escape code.
- An End Of Block code is encountered; Depending on the `intra_vlc_format` this is a 2 or 4 bits code

Apart from the special cases mentioned above, the Entropy Decoding process is regular. This process forms along with other processes the MPEG-2 decoding process as depicted in Figure 2-2.

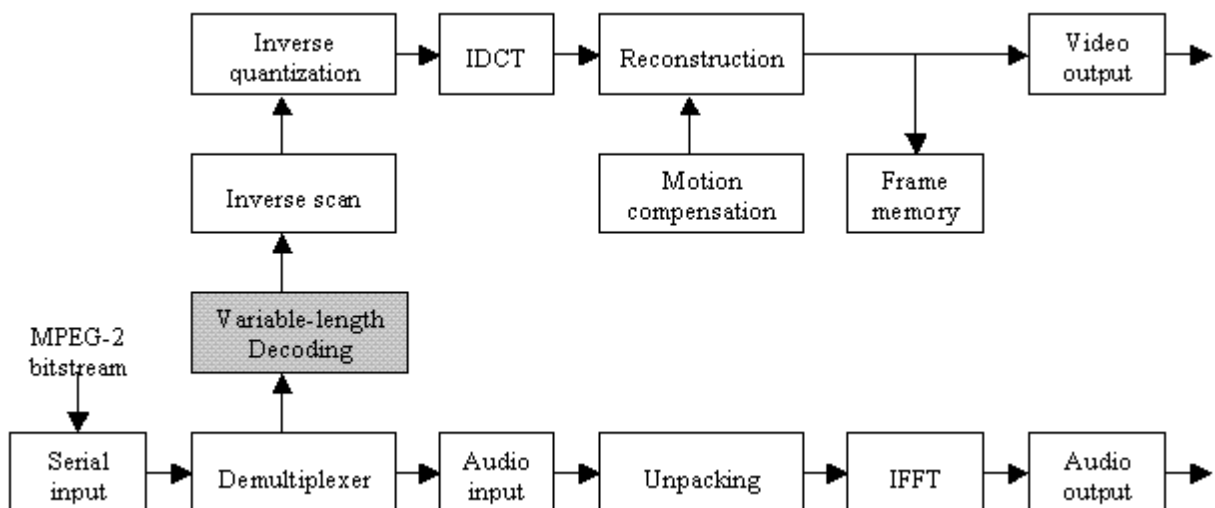


Figure 2.2: A schematic overview of the MPEG-2 decoding process

A MPEG-2 decoder gets a video bit-stream as an input. This bit-stream is hierarchically organized into layers [17]. These layers are in chronological order:

- The incoming MPEG-2 bit-stream
- A group of pictures
- The picture itself

- A slice from a picture
- A macroblock in this slice
- A block from the macroblock

These terms are frequently referred to throughout the decoding process. The picture of the organization shows how to place and visualize these hierarchical layers.

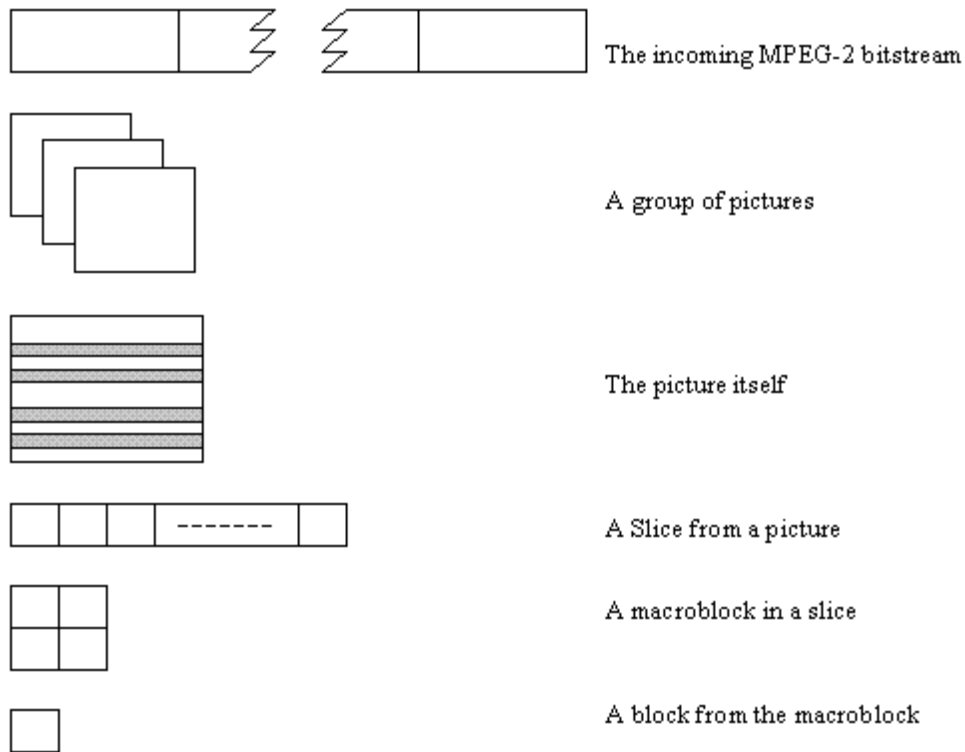


Figure 2.3: Hierarchical organization of the MPEG-2 video bit-stream

Headers separate all the different parts of the incoming bit-stream. These headers are byte-aligned start-codes [17]. These start-codes are used both to identify parts of the stream and to allow random access into the video-stream. The random access ability is vital for parallelization techniques. This is explained in chapter 4.

2.5 Theoretical Background Conclusion

This chapter has discussed the methods used in this thesis. These methods will be used on the processors mentioned. The explained methods are:

- Entropy Decoding
- Run Length Decoding
- Variable-Length Decoding

Entropy Decoding is an essentially sequential task and lies on the critical path of MPEG-2 decoding. Therefore this is an intricate function on processors that benefit from parallelism. Entropy Decoding being a part of MPEG-2 Decoding decodes picture blocks. This is executed according to the specified *intra_vlc_format* in the *macro_block_header*. The specification of *intra_vlc_format* is given in Table 2-6. This shows that depending on this bit a Lookup Table is chosen to decode the following situations:

- Non-Intra blocks
- Intra blocks starting with a Chrominance coefficient followed by
 - Coded symbols to be looked up in table B14
 - Coded symbols to be looked up in table B15
- Intra blocks starting with a Luminance coefficient followed by
 - Coded symbols to be looked up in table B14
 - Coded symbols to be looked up in table B15

Entropy Decoding itself can be split into two parts. These parts are strictly sequential because the result of Run Length Decoding depends on the result of Variable Length Decoding. Run Length Coding as being the inverse operation of Run Length Decoding is a compression technique that like others decreases redundancy in data representation. Variable Length Coding as being the inverse operation of Variable Length Decoding is based on the Huffman coding principle [8].

TriMedia & SpaceCAKE

TriMedia and SpaceCAKE are processors on which the previously discussed methods are tested. Also the optimization steps taken are done for TriMedia and SpaceCAKE. This chapter explains the capabilities and properties of both processors.

Section 3.1 explains the TriMedia architecture and its tool-set. The Very Long Instruction Word is emphasized, because the VLIW principle forms the basis of all optimization steps taken. The used SpaceCAKE architecture is shown in section 3.2.

3.1 TriMedia Architecture and Tool-set

The history of TriMedia starts in 1987 with a LIFE-1 VLIW processor designed by Junien Labrousse and Gert Slavenburg [15] [13]. The work on this processor continues at Philips Research. The first TM1000 processor appeared in 1996 [15] [13]. Further development on this processor has resulted in the TM1300 and TM1500 [16]. A TriMedia processor is intended to be a multi-standard video, audio and graphics processor that is embedded in consumer electronics. In addition, a TriMedia processor is also usable as a master CPU in a stand-alone multimedia PCI-bus-based system. In this case, it boots from an attached serial EEPROM. The application software is brought in from a PCI bus attached ROM or from a peripheral device.

The TriMedia/CPU32 consists of a set of coprocessors and the TM32 core. The core is a 32-bit high performance five-issue slot Very Long Instruction Word (VLIW) CPU. This core is capable of executing a maximum of twenty-seven operations per cycle. The sustained execution rate is about five operations per cycle for tuned applications. This performance depends heavily on the kind of operations and possibilities for parallelism. The TM32 core is equipped with a special register file. This register file has ten read and five write ports available for a Very Long Instruction Word. This gives opportunities for executing a single instruction on multiple data (SIMD). In such way parallel vector operations can be performed using custom operations for the TM32. Another principle that benefits from the organization of the register file is the multiple instructions on multiple data (MIMD) principle. This gives opportunities to execute up to five independent operations simultaneously on the five-issue slot VLIW TM32 core. These operations can target up to 27 functional units in the processor that include i.e.:

- Floating point operations; The TM32 is capable of single precision arithmetic on floating point variables.
- Integer operations; The architecture of the TM32 supports the notion of signed

and unsigned integers and their arithmetic. Signed integers use the standard two's-complement representation.

- Data-parallel DSP-like operations; These are custom operations that perform clipped arithmetic on vectors of two words or on integers.

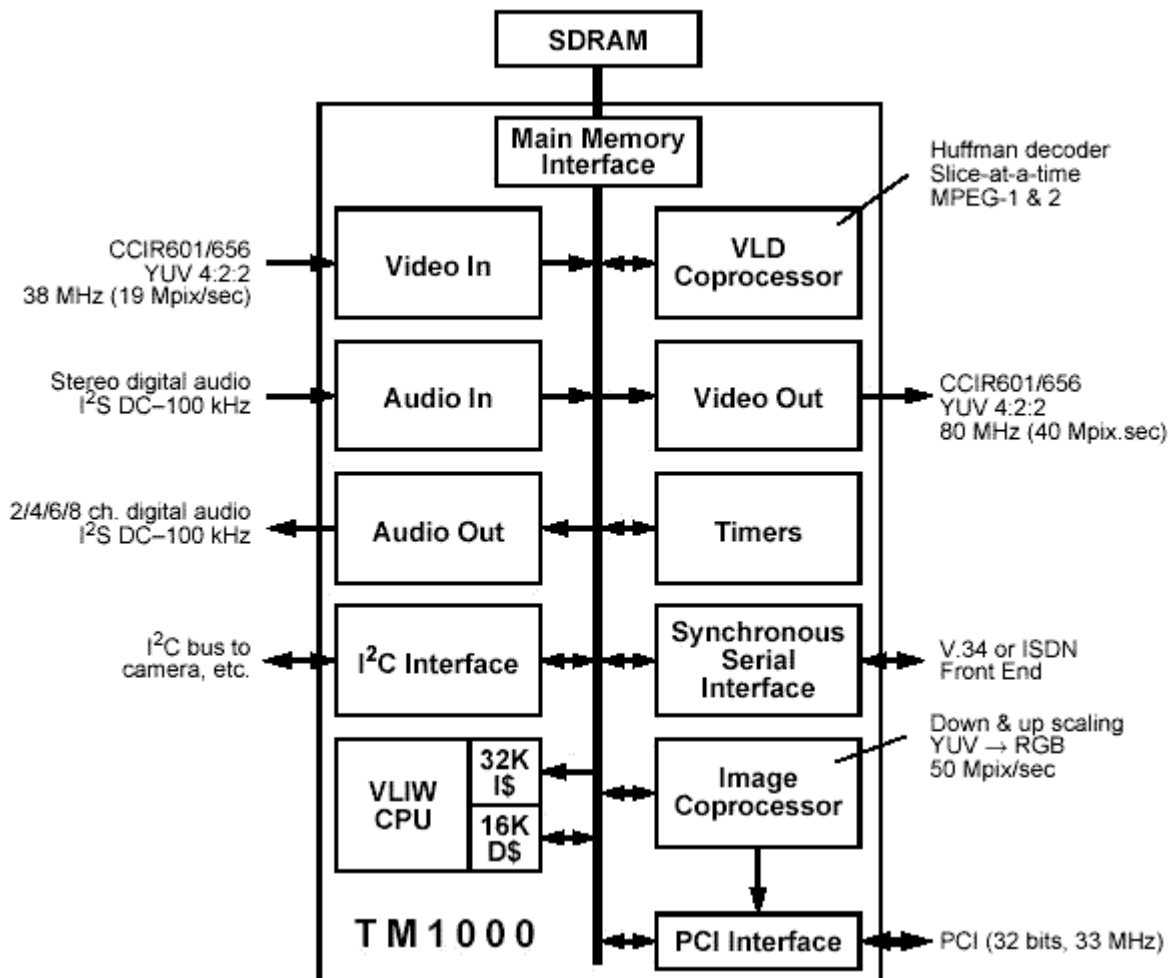


Figure 3.1: TM1000 block diagram An example of the TriMedia

The instruction set of the TriMedia processor is optimized for processing audio, video, and 3D graphics media. This includes the mentioned SIMD custom operations for eight and sixteen bit data types. Furthermore, it includes a full complement of 32 bit IEEE compatible floating-point operations. The result of these operations is already usable in the next cycle. An example of custom multimedia operations performed on a TM32 core is byte selection. The source code of signed byte selection in plain C is:

```
Byte0 = ((signed_integer << 24) >> 24);
Byte1 = ((signed_integer << 16) >> 24);
Byte2 = ((signed_integer << 8) >> 24);
Byte3 = signed_integer >> 24 ;
```

The equivalent of these operations in custom operations is:

```
Byte0 = sbytesel(signed_integer, 0);
Byte1 = sbytesel(signed_integer, 1);
Byte2 = sbytesel(signed_integer, 2);
Byte3 = sbytesel(signed_integer, 3);
```

This example shows that selecting a byte from a four-byte vector can always be done in one cycle with custom multimedia operations. The "s" in front of "sbytesel" shows that a signed byte is extracted as in the plain C example. This multimedia operation is also possible for unsigned bytes using the "ubytesel" function. The plain C version of this code needs two cycles in most cases for the same operation. This shows the advantage of multimedia operations in a simple example.

The TM32 core coordinates all on chip activities. In addition to implementing the non-trivial parts of the programs it runs it also runs a small real-time OS kernel. This kernel is driven by interrupts from the coprocessors.

A variety of coprocessors is located around the core. These processors are specialized in tasks like:

- Variable Length Decoding (for MPEG-1 and MPEG-2, Slice at a time).
- Image processing for down/up scaling and YUV \rightarrow RGB.
- Transfer of a video-signal in and out (YUV 4:2:2).
- Transfer of an audio-signal in and out (Digital audio).
- Communication through a synchronous serial interface (V.34 or ISDN).
- Communication through a PCI interface (32 bits, 33 MHz).
- Communication through an IC interface.
- Timing.

The coprocessors are connected through a high-bandwidth internal bus. The VLIW core communicates with a coprocessor through the main memory. Each coprocessor is seen as a memory location. The VLIW core writes to and reads from the memory location to communicate with a coprocessor. In this way the control registers of the coprocessor appear to be part of the main memory.

The TriMedia processor supports popular multimedia standards like MPEG-1 and MPEG-2. This gives capabilities for implementing a variety of multimedia algorithms, which are open or proprietary. This is applied as an accelerator in a PC environment, or as the sole CPU in a stand-alone system.

The TM1000 relieves a general purpose CPU in a PC environment. When the TM1000 was introduced PC CPU's were not yet capable of dealing with real-time compression and decompression of high quality audio- and video-streams. At this point the TM1000 performs the task of an accelerator in a PC system. This makes it possible to provide live high quality video and audio entertainment without sacrificing the responsiveness of the PC system. For example playing the compressed MPEG-2 video files on a Pentium I CPU gives a high processor load. Adding the TM1000 to this system assures the responsiveness of the system. The TM1000 has more advantages in addition to acting like a fixed function multimedia accelerator, namely:

- A special-purpose embedded solution with low cost and chip count.
- A general-purpose processor with its reprogram ability.

Very Long Instruction Word (VLIW)

A very important part in the TriMedia processor is its core based on the Very Long Instruction Word (VLIW) principle. All operations in a very long instruction word are executed concurrently [7]. In this way Instruction Level Parallelism (ILP) can be exploited when present. This results in parallelism without the cost of multiple processors in a system. This maximizes processor throughput at the lowest cost also in comparison to superscalar CPU's. VLIW architectures have performance exceeding that of superscalar general-purpose CPU's without the complexity of a superscalar implementation [15] [13].

Executing five operations in one instruction needs special hardware. There are 27 functional units available containing all operations like branches, floating point and several others. All functional units have access to the same set of registers via a read and wire crossbar. This gives an organization as shown in Figure 3-2.

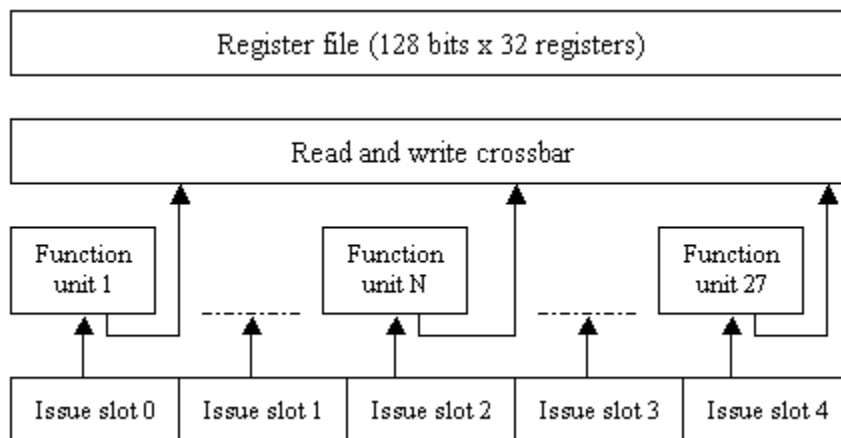


Figure 3.2: TM1000 DSPCPU-organization

Scheduling the instructions in the organization as shown in Figure 3-2 is done at compile time. Doing this at compile time has the advantage of a lower hardware cost. There will be no hardware needed for runtime branch prediction or scheduling. Scheduling can be done very carefully and thoroughly when it is done at compile time. On the other hand scheduling at compile time has two disadvantages:

- It takes a lot of time, much more than a real-time scheduler.
- The compiler/scheduler becomes very complex.

When options like loop-unrolling and grafting are used, scheduling at compile time results in good performance. The optimization level of the scheduler can be set manually before compilation. This allows an optimization level of zero to five with the following properties:

0. No optimization
1. Local optimization only, variables in stack. (Per basic block)
2. Local optimization only, variables in registers. (Per decision tree)
3. Global optimizations
4. Inter procedural analysis and inlining
5. More extensive inlining and global optimizations

Table 3.1: Mapping from optimization level to optimizations [2]

		Optimization Level					
Optimization	Variable	-O0	-O1	-O2	-O3	-O4	-O5
Alias Analysis	-Xalias	0	1	3	3	4	4
Call Modification Analysis	-Xcallmod	0	1	1	1	2	2
Constant Propagation	-Xconstp	0	0	2	2	2	2
Copy Propagation	-Xcopyp	0	0	2	2	2	2
Loop Forward Code Motion	-Xfem	0	0	1	2	2	2
Control Flow Analysis	-Xflow	0	0	1	1	1	1
Inlining Level	-Xinlev	0	0	0	0	1	1
"Main" Optimizations	-Xmopt	0	1	3	4	4	4
Register Allocation	-Xreg	0	0	1	1	3	3
Loop-Unrolling	-Xunroll	0	0	1	1	1	1
"Extra" Optimizations	-Xxopt	0	0	2	2	3	5
Expression Reordering	-Xzone	0	0	1	1	1	1

Table 3-1 shows the mapping from optimization level to optimizations. The default optimization level is -O3. Other optimizations are possible with profiling and grafting

utilities. Profiling works through a compile-profile-recompile cycle of performance tuning [1]. First the program is compiled using the compiler driver with an option to make it write profiling information to a file. Next, the program execution is simulated using the machine-level simulator. This results in a file with information about the execution of the program. This file is used when recompiling the program with the profile driven option. In this way optimizations like loop-unrolling take place. Grafting based on profile information increases parallelism within decision trees. This technique replaces any jump with a copy of the destination tree and thus "grows" larger decision trees [1]. As a result the program size increases, as does its performance.

3.2 SpaceCAKE

A SpaceCAKE instance is a tile in a configurable homogeneous multiprocessor network [12]. It is a high-end multimedia processor with an easy to program multiprocessor architecture. A possible SpaceCAKE instance consists of eight TriMedia cores and one MIPS core supported by memory banks. Almost all communication between the cores in the tile runs over a configurable interconnection network. This communication is taken care of by the router. In some cases this type of communication is not practical. At this point a bridge to shared memory is used.

A possible SpaceCAKE architecture is shown in Figure 3-3.

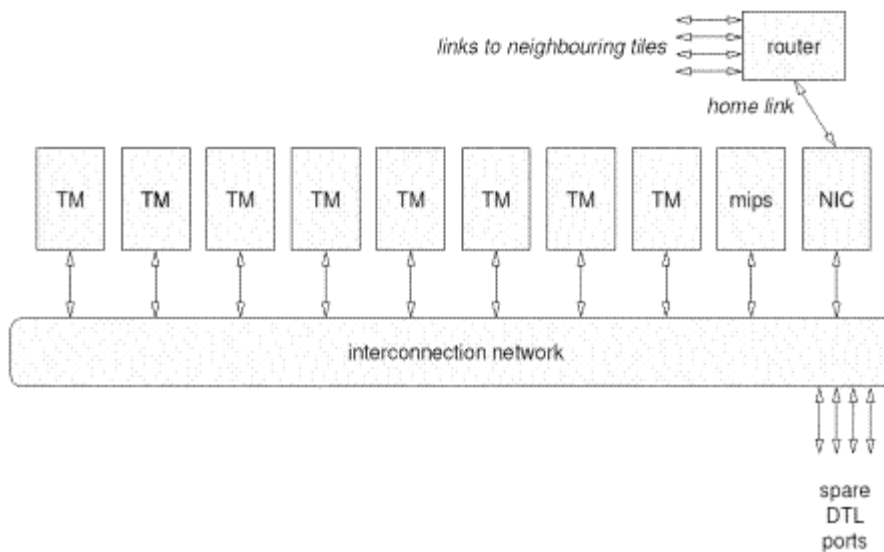


Figure 3.3: A Typical architecture of a SpaceCAKE instance

The advantage of an architecture as shown in Figure 3-3 is its intended speedup for multimedia applications. This speedup can be regarded as linearly dependent on the number of allocated CPU's with two conditions: this is subject to overhead and

the exploitability of parallelism. In this way a SpaceCAKE instance benefits from two different levels of parallelism:

- Parallelism inside a TriMedia core based on the VLIW principle instruction level parallelism (ILP)
- Parallelism created between a number of TriMedia cores data level parallelism (DLP)

A SpaceCAKE instance supports a TM32 core with more TM32 cores instead of coprocessors. As seen in Table 1-1 there is a trade off between a hardware and a software solution. The TM1000 as depicted in Figure 4-1 is supported by co processors and is therefore a hardware solution. The software solution is SpaceCAKE, where the support comes from programmable TM32 cores.

3.3 TriMedia & SpaceCAKE Conclusion

The TriMedia/CPU32 is a processor supported by a set of coprocessors as depicted in Figure 3-1. The core of the processor named the TM32 core is based on the Very Long Instruction Word principle. This allows for five independent instructions to be executed simultaneously. This gives opportunities for executing single instructions on multiple data (SIMD). This is useful for parallel vector operations. In addition to this also multiple instructions on multiple data is possible due to the register organization and 27 available functional units. All these operations are scheduled at compile time. This has the advantage of a lower cost in hardware and the disadvantage of a time consuming compiling stage. When compiling a set of optimization options are set as shown in Table 3-1.

SpaceCAKE is a cluster of processors that consists of a set of TM32 cores and MIPS cores. These are connected through a network to make it a tile. This tile is a configurable homogeneous multiprocessor network [6]. This tile gives opportunities for two levels of parallelism namely:

- Parallelism inside a TriMedia core based on the VLIW principle instruction level parallelism (ILP)
- Parallelism created between a number of TriMedia cores data level parallelism (DLP)

This approach is intended to benefit from the hard-/software solution trade off as shown in Table 1-1.

Stand-alone Entropy Decoding on TM32

4

As shown in section 1.2, Entropy Decoding is on the critical path of MPEG-2 decoding. An efficient implementation of Entropy Decoding on the TriMedia32 core is needed. This implementation is based on The Entropy Decoding source code written for TriMedia/CPU64. However this implementation is not straightforward applicable for TriMedia/CPU32. Therefore a porting step is taken. From this source code five optimization steps are done.

Section 4.1 explains porting the source code written for TriMedia/CPU64 to TriMedia/CPU32. This source code is optimized by re-ordering and compressing the data it needs from the Lookup-Table which is explained in section 4.2. After this the data from the incoming bit-stream is analyzed (section 4.3). From this a set of unique cases emerges that comes forward in section 4.4. Section 4.5 continues by removing branches. This is exploited further in Section 4.6 by executing pre-loads.

4.1 Porting TM64 specific code for Entropy Decoding

Entropy Decoding is an essentially sequential task in which only little parallelism can be created. As explained in section 2.4, it consists of VLD followed by RLD, where RLD relies on the results of VLD. I.e. RLD and VLD can run independently as long as RLD is preceded by VLD. The goal is to optimize Entropy Decoding using properties such as the VLD - RLD relation and using optimization techniques like code pipelining, data re-ordering and pre-loading. The starting point of these optimization techniques is the Entropy Decoder written for the TM64 core. This decoder exploits some parallelism (ILP) but specific for the TM64 core. The TM64 core differs from the TM32 core in its register and cache size that both are doubled in the TM64 core. The developed source code is therefore not directly usable for a TM32 core, but can be seen as a good starting point. All methods and algorithms written for the TM64 core code can be ported to be useable on a TM32 core. Porting the source code involves several adaptations. Compiling the source code written for the TM64 core with a compiler for the TM32 core gives several expected errors. These errors arise in two categories:

- Errors caused by a decrease in register size
 - Unknown data types
 - Data size exceeded
 - Unpredictable shift operation
- Errors caused by unavailable operations

- Unspecified operation, custom operation not known for the TM32 core
- Operation with too many arguments, super operation not available for the TM32 core

Each unknown data type is matched with a possible candidate for the TM32 core. Since the TM32 core has only a few data types of its own, generally plain C data types are chosen to replace the TM64 core data types. Examples of these replacements are:

```
uint8 run;
int16 level = 0;
uint8 Escape;
uint8 Valid_Decode;
uint16 table_offset;
uint16 lookup_address_width;
uint32 lookup_address;
```

These variable declarations are ported to the TM32 core and become, shown in the same order:

```
unsigned int run;
signed int level = 0;
unsigned int Escape;
unsigned int Valid_Decode;
unsigned int table_offset;
unsigned int lookup_address_width;
unsigned int lookup_address;
```

Adaptations as illustrated finally result in a source code in plain C that is transferable to many platforms. This gives the possibility to do a fair comparison in test and performance results for different platforms.

When looking at the Lookup Table used for Entropy Decoding on the TM64 core, it shows that data is treated in a different way. The data cannot be seen bit by bit as from the incoming bit-stream, but it is organized byte by byte. The bytes stored in the Lookup Table are:

- Level
- Run
- Valid decode
- Escape
- Code Length
- Lookup address width
- Table offset
- End of Block

This data is stored byte aligned on 64 bits. With a special function "bytesel" unsigned data is retrieved out of the data from the Lookup Table. This approach is also taken in the source code for the TM32 core. The following example illustrates how this adaptation is carried out for extracting the level value.

```
vec64ub retrieved_vec64ub;
byte level = bytesel(retrieved_vec64ub, DCT_COEFFICIENT_LEVEL_FIELD);
level = (level << 24) >> 24;    /* sign extension */
```

The equivalent source code for the TM32 core becomes:

```
signed int retrieved_vec64ub_1;
unsigned int retrieved_vec64ub_2;
signed int level = retrieved_vec64ub >> 24;
```

Here adaptations are made for the unknown data type vec64ub. In addition, the function "bytesel" is replaced. The level field is placed at the eight most significant bit positions. This allows for "level" to be extracted with the correct sign in a single instruction.

The example above only shows source code for the level byte. As said, the other seven bytes are stored byte aligned in the Lookup Table. This total of eight bytes forms a vector that is stored in a one-dimensional array. This array forms a Lookup Table i.e. the B14 Lookup Table. This table is split into eight smaller parts as defined by the "table_offset" byte [14]. These parts hold the same order as the rows in the MPEG standard to maintain readability. The number of address bits for each table is related to the maximum length of the variable-length codes. This is shown in Table 4-1

Table 4.1: Number of address lines, size and offset for each Lookup Table

Table	No. of address lines (lookup_address_width)	Size 64-bit words	table_offset
first	8	$2^8 = 256$	0x000
second	8	$2^8 = 256$	0x100
third	4	$2^4 = 16$	0x200
fourth	4	$2^4 = 16$	0x210
fifth	5	$2^5 = 32$	0x220
sixth	7	$2^7 = 128$	0x240
seventh	5	$2^5 = 32$	0x2c0
eighth	5	$2^5 = 32$	0x2e0

Table 4-1 shows a total of 768 64-bit words, which means 6KB of storage space. The last column shows the "table_offset" that always ends on a "0". This gives the opportunity to reduce the storage space needed for this value when cutting of this "0" but memorizing that it has been done when using the value. In such way the number of bits needed to store the "table_offset" value decreases from 10 bits to 6.

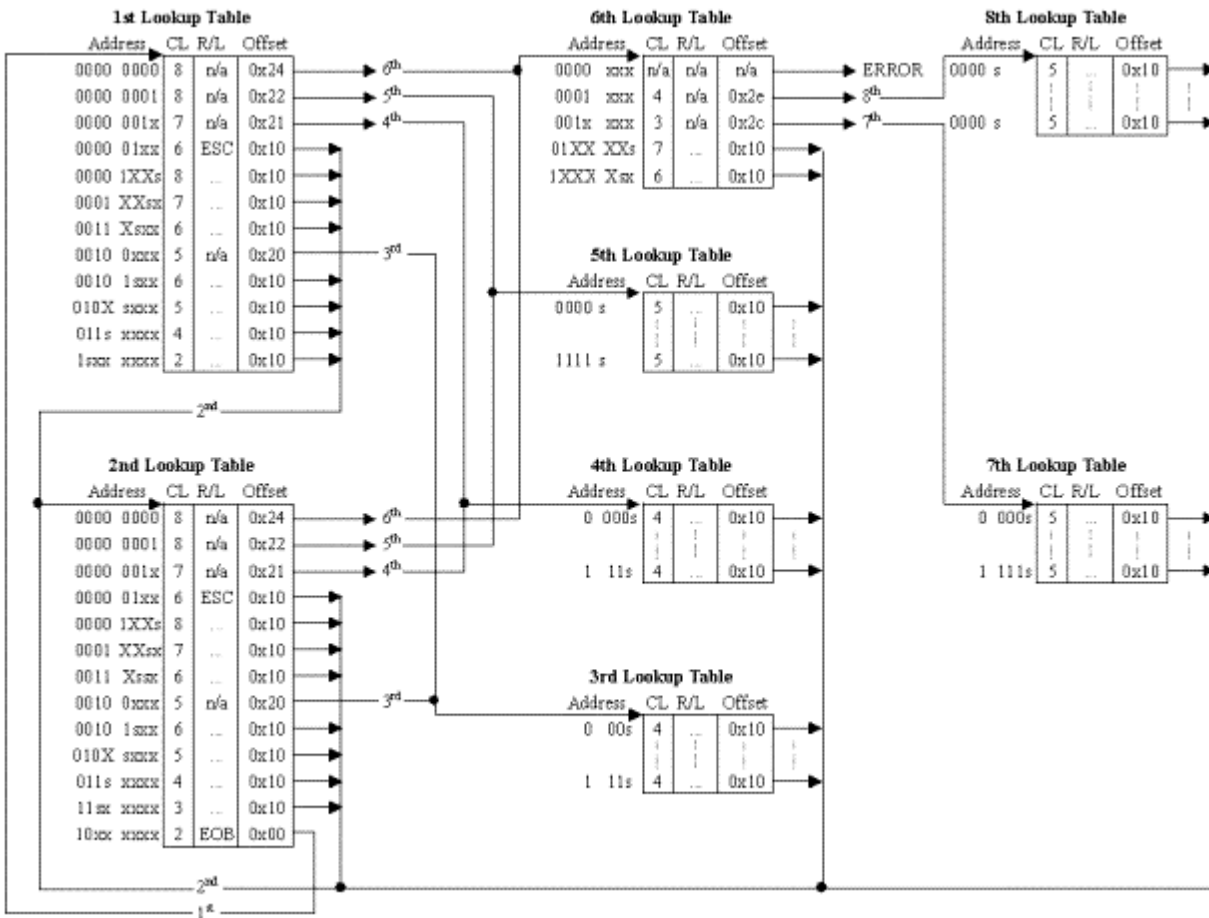


Figure 4.1: The flowchart of the variable-length decoding procedure

The decoding procedure is exemplified in Figure 4-1 [14]. Suppose the following string is decoded: "1000000000011000110". The table_offset is initialized to 0 and the lookup_address_width is set to 8, that is the first table is being pointed to as seen from Table 5-1. This means that the first eight bits of the string, i.e. "10000000", are regarded as address into the first table. The following values are retrieved: code_length = 2, run = 0, level = 1 table_offset = 0x100 and lookup_address_width = 8. This means that the second table is accessed in the second iteration. After shifting out the two bits decoded in the previous iteration, the next eight bits, i.e. "00000000", are the address into the second table. This lookup results in code_length = 8, table_offset = 0x240 and lookup_address_width = 7. In this case the sixth table is accessed in the next iteration. In addition, there is no run-level pair detected in this iteration step. This pair is not always generated in each iteration step. At this moment the acc_code_L = 10. Therefore, the first 10 bits from the stream are shifted out. The next seven bits, i.e. "0011000" becomes the address in the sixth table. Again no valid run-level pair is detected. The code_length = 3, table_offset = 0x2c0 and lookup_address_width = 5. With this result the seventh

table is accessed. The resulting `acc_code_L` has become 13 resulting in shifting these bits out and leaving the next five bits, i.e. "10001" to become the address in the seventh table. This lookup results in `code_length = 5`, `run = 0`, `level = -32`, `lookup_address_width = 8`, `table_offset = 0x100`; this bypasses the first table. All subsequent coefficients of the emerging block will use only the second to eighth table. In the next step `acc_code_L = 18`, this results in the eight bit lookup address "10xxxxxx". With this address an `end_of_block` symbol is encountered. The `table_offset` becomes 0x000 so the next block is decoded starting from the first table.

Errors involving exceeded data size need to be examined individually. Splitting a variable in two parts and joining them in a structure seems to be the easiest solution to tackle the problem. This will cause a lot of overhead in data management. Avoiding this overhead where possible is more attractive. Therefore the data that comes from the incoming bit-stream is put in the same two variables as in the source code written for the TM64 core. However, the size of these variables is halved. This results in halving the amount of bits available to the decoding process. The decoding process is not limited in its functionality since MPEG-2 decoding requires at most 24 bits from the incoming bit-stream to perform one decoding step. With the availability of 32 bit positions for this data from the incoming bit-stream the requirement is met. However, tightening the amount of new data available to the decoder will introduce more checks for refreshment of the available data. This gives in two cases disadvantages to consider.

- In case of a structure of two joined integers, data refreshment and data retrieval become more complex.
- Reducing the amount of available data affects data refreshment.

The second case has fewer disadvantages compared to the first. Therefore, the maximum number of fresh available data bits is reduced from 64 to 32. This is illustrated with source code at the end of this section.

This reduction of available data bits affects the data management carried out with shift operations. When shifting over a larger amount than 31 positions, the result is not defined on a 32-bit processor. Possible shift operations are:

```
data_integer >> 0;      /* 1 */
data_integer >> 26;     /* 2 */
data_integer >> 32;     /* 3 */
data_integer >> 58;     /* 4 */
```

When the "data_integer" is defined as a 64-bit long integer on the TM64 core no problems occur with these shift operations. The TM32 core gives non pre-defined results on the third and fourth instruction. When the TM32 core encounters these instructions, it shows only to take the last five bits of the parameter into account. This results in an implicit modulo 32 operation on the parameter. However, this is not robust programming because of implicit operations and therefore needs to be avoided. This calls for the need of a new data refreshment structure. In this approach the data refreshment takes place after the return from each function that has processed data from the incoming bit-stream. Only in this way it can be guaranteed that the next executed function has enough data from the bit-stream to work with.

In addition to errors related to the decreased register size, errors related to unavailable operations occur as well. This is mainly caused by the smaller instruction set available on a TM32 core compared to the TM64 core. The TM64 core is capable of executing so called super operations. These operations use two VLIW issue slots instead of one. This results in the availability of twice the number of registers for input and output data. Super-operations are especially useful when data from two registers needs to be combined where an input parameter determines the output data. This is shown in the following source code:

```
uint64_bitfunshift( &VLC_Ry, &VLC_benign,
                   MSDW_VLC_string_chunk,
                   LSDW_VLC_string_chunk,
                   acc_code_L);
```

This source code shows the `uint64_bitfunshift` super operation. This operation has the last three parameters as input parameters and the first two as output parameters. It therefore needs five registers to execute. The result of this operation is:

1. The concatenation of the 64 minus "acc_code_L" most significant bits from the "LSDW_VLC_string_chunk" and the "acc_code_L" bits from the "MSDW_VLC_string_chunk"
2. The "acc_code_L" least significant bits from the "LSDW_VLC_string_chunk"

The first result is stored in "&VLC_Ry" and the second is stored in "&VLC.benign". In this way a set of new bits from the incoming bit-stream is selected to get decoded through the first result. The new bits that still need to be decoded are stored as well through the second result. The TM32 core has no super operations. This operation therefore needs to be rewritten for this core. This results in the following set of instructions:

```
if (acc_code_L == 0)
    VLC_Ry = MSDW_VLC_string_chunk
else
    VLC_Ry = ((MSDW_VLC_string_chunk << acc_code_L) |
              (LSDW_VLC_string_chunk >> (32 - acc_code_L)))
```

The if-else combination is needed to avoid the situation in which data from the "MSDW_VLC_string_chunk" is overwritten with data from the "LSDW_VLC_string_chunk". The remainder of the set of instructions is functionally identical to the super operation of the TM64 core. However, the TM32 core equivalent needs four cycles instead of one for this remainder. Furthermore it works only with an "acc_code_L" ranging until 32 instead of 63 for the TM64 core.

4.2 Re-ordering data organization in the Lookup Table

As said in section 4.1, the data retrieved from the lookup table does not fit in one variable anymore. The lookup table stores information about eight parameters. All of these are

byte aligned. This results in two variables on the TM32 core and one on the TM64 core. Working with a pair of variables instead of one results in more overhead when reading from or writing to this variable. Therefore performance can be gained when the parameters are stored in one variable on the TM32 core. When looking at the space in bits needed for this action the result is:

- 8 bits taken for level
- 6 bits taken for run
- 1 bit taken for valid decode
- 1 bit taken for escape code
- 4 bits taken for code length
- 4 bits taken for lookup address width
- 6 bits taken for table offset
- 1 bit taken for end of block

The sum of the line-up above is 31 bits. Spreading these parameters over 64 bit positions results in 33 unused positions. Where this is of no harm on the TM64 core, the TM32 core does not benefit of this arrangement of bits. When looking at the consequences of rearranging the data in the lookup table a list of situations can be made for a TM32 core as done in Table 4-2.

Table 4.2: Overview of data compression effects on the number of cycles needed for involved operations

64-bit results from lookup table	32-bit results from lookup table
<ul style="list-style-type: none"> • Two variables for parameters <ul style="list-style-type: none"> ◦ Takes 2 times n cycles every load operation 	<ul style="list-style-type: none"> • One variable for parameters <ul style="list-style-type: none"> ◦ Takes 1 times n cycles every load operation
<ul style="list-style-type: none"> • Retrieval of a parameter <ul style="list-style-type: none"> ◦ Takes 3 cycles for level <ul style="list-style-type: none"> ▪ Selecting without the need of masking (1) ▪ Sign extension (2) 	<ul style="list-style-type: none"> • Retrieval of a parameter <ul style="list-style-type: none"> ◦ Takes 1 cycle for level <ul style="list-style-type: none"> ▪ Signed shift without the need of masking
<ul style="list-style-type: none"> ◦ Takes 2 cycles for other parameters <ul style="list-style-type: none"> ▪ Shifting ▪ Masking 	<ul style="list-style-type: none"> ◦ Takes 2 cycles for other parameters <ul style="list-style-type: none"> ▪ Shifting ▪ Masking
<p>In these cases n usually is equal to three for a load or store operation</p>	

Comparing the two situations shows that getting 32-bit results from the Lookup Table only has benefits. However, this needs careful organization of the arrangement of parameters in the variable to benefit. When level has to be extracted in one cycle with only a signed shift operation, it needs to be on the eight most significant bit positions.

Furthermore an analysis is made about the frequency of occurring single bit parameters. The most frequent occurring are placed at the least significant bit position. This gives the largest benefit from using a guarded operation when checking this bit. The Valid Decode bit is not a signal bit. It is used as a value in choosing the position in the matrix that is filled in with levels as shown in the source code example below.

```
nz_coef_pos_ZZ += run;
nz_coef_pos = invZZ_table[nz_coef_pos_ZZ];
unpacked_8x8_matrix[nz_coef_pos] = level;
nz_coef_pos_ZZ += Valid Decode;
```

This leaves two bits that are signal bits, End of Block and Escape. These are used in the following source code example.

```
if ((acc_code_L < 32) && !Escape)
    goto cont;

if (EOB)
    break;
/* EOB (or error if not-valid is also set) */
```

The Escape bit is always set when the End Of Block bit is set in the implementation of the used Lookup Table. This leads to the Escape bit being the most frequently checked signal bit. Therefore this bit is chosen to take the least significant bit position.

The discussed arguments give the result depicted in Figure 4-2. The bottom line shows the bit positions.

Level								Empty																Esc							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

Figure 4.2: Filling of crucial bit positions

Filling the other positions is done byte wise. This is according to the arrangement chosen for the TM64 core lookup table. The byte select function is also available for the TM32 core. Compared to a straightforward shift and/or mask, the byte select function also takes one cycle to execute. Therefore no difference in performance can be seen choosing either of both. The advantage of using the byte select function is getting a more clear view over the data treatment in the second cycle when extracting a parameter. The disadvantage is the third cycle that is needed to extract the Valid Decode parameter.

Another disadvantage is that the byte select function is not a plain C function. This makes the code less portable to other platforms. The complete filling of the integer is shown in Figure 4-3.

Level								Run								VD	EOB	Code Length				Lookup addr. width				Table offset								Esc
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00			

VD = Valid Decode
EOB = EndOfBlock

Figure 4.3: The complete filling of bit positions

The order of checking the VD, EOB and ESC positions forms an important factor in the performance of the Entropy Decoder. When the least frequent occurring flag is checked first, this check often results in the conclusion that this flag was not set. After this check more checks on other flags need to be executed. In this way a lot of branch instructions are realized that consume a lot of instruction cycles. Therefore the most frequently set flag needs to be checked first. What flag this is can be concluded from the properties of a MPEG-2 bit-stream. The percentages of occurrence of a flag are shown in Table 4-3.

Table 4.3: Overview of occurrence of the VD, EOB and ESC flags

Stream	Valid Decode	EOB	ESC
bat_327_334	85%	14%	0,8%
popplen	88%	12%	0,9%
queen_2248_30	71%	28%	0,4%
sarnoff2	85%	14%	0,5%
tek3	86%	13%	0,8%
tennis	82%	14%	4,2%
tilcheer	86%	13%	0,3%

Table 4-3 shows that the Valid Decode flag is set most often and therefore needs to be checked first. This will minimize the number of branches to find out what flag is set. The EOB flag becomes the next to be checked. This leaves the ESC flag to be checked. However, this flag actually does not need to be checked since the other options have already been ruled out. This gives little optimization since the occurrence of the ESC flag is low. Nevertheless, it is an optimization of the source code.

4.3 Entropy Decoding Organization

The Entropy Decoder built up to now is used for future reference in this thesis. The organization of its blocks and order of operations is explained.

1. The Entropy Decoder starts with a set of declarations and initializations.
2. With this result a loop starts that processes all blocks. The number of blocks is set at initialization time.
3. Within the loop over the number of blocks initial values are set to decode each first symbol from a block.
4. At that point an infinite loop starts. This loop processes data from the incoming bit-stream as long as no EOB flag is decoded. Throughout this process run-level pairs are generated according to a system similar to the example in Figure 4-1.
5. The goto statement at the end of the loop clarifies the loop structure.
6. When an EOB flag is encountered, the process breaks out of the infinite loop and starts decoding a new block as long as there are blocks present.
7. When no block is present, de-initializations take place and the Entropy Decoding process is ended.

These steps taken are visualized in Figure 4-4.

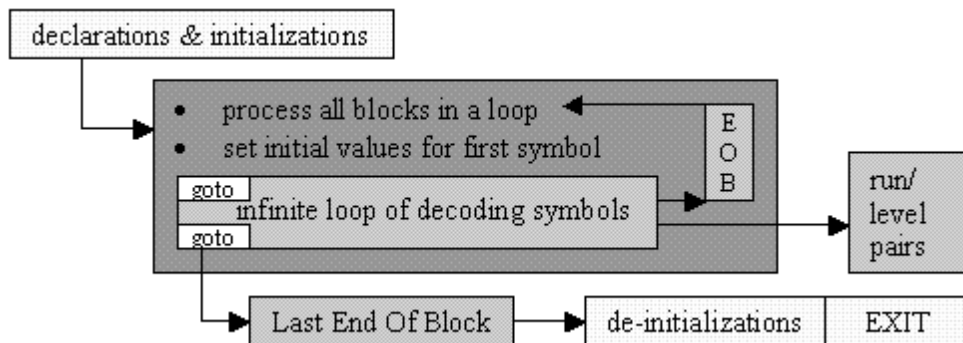


Figure 4.4: Entropy Decoding organization scheme

4.4 Identification of unique cases

The Entropy Decoder has some unique cases that need to be executed before the "infinite" loop in Entropy Decoding starts. The loop consists mainly of sequential instructions. This structure will result in a source code with a high level of possible parallelism. Each branch in the source code needs about three cycles to be evaluated and executed.

These three cycles are very costly since very little other instructions can be scheduled to take place in parallel with the execution of the branch. Therefore the branches in the "infinite" loop of Entropy Decoding need to be placed at the end. When a branch is needed, the guarded operation is preferred above a common if statement. A guarded operation does not introduce bubbles in the processor pipeline like branches do. In addition, the scheduler for the TM32 core is sensitive to this kind of programming, and will be able to optimize the execution of this operation. This saves a lot of cycles compared to a traditional if case in a branch instruction.

At the starting point of the Entropy Decoding process all initializations take place. This is nothing new at all, but the initialization causes a unique case to be considered. After the initialization of the Lookup Table and variables has taken place, no information is stored in the variables that are used for the incoming bit-stream. This is the only case in the whole process where this appears. This is because not only the variables used for the incoming bit-stream are empty, but also the number of decoded bits is zero. The last case will never appear again in the decoding process. Treating the unique case within the loop gives a less performing source code. The instructions are therefore placed in front of the loop. Furthermore the data management of the incoming bit-stream gives a special shift operation. As discussed before, shift operations on 32-bit integers are strictly defined when the shift parameter lies between 0 and 31, with both limits included. Shifting over zero positions is not considered since this shift operation is useless. Shifting over 32 positions is not strictly defined. One processor might give an empty integer as a result; the other might leave the integer as it is. This problem can be solved with three types of solutions. These are:

- Smart non-conditional shifting
- If-else conditional shifting
- Question mark conditional shifting

Smart non-conditional shifting

To be sure about the merged result of a shift over 32 positions, a double shift is possible. The integer is shifted over 1 position at first. After this shift, the integer is shifted over the remaining 31 positions. Both shift operations are strictly defined now, so a shift over 32 positions has become predictable. The method introduced above, can be used in more cases. The goal of a shift is to introduce a number of zeros in an integer in this case. This number is defined by the shift parameter. An integer has 32 positions. Introducing no zeros in an integer is in this case considered as being useless. On the other hand introducing 32 zeros in an integer is regarded as being a possible result. Therefore, the shift parameter lies between 1 and 32 where the limits are included. This means that the integer is definitely shifted over 1 position. This operation is executed in all cases. In the same instruction line the shift over the remainder of positions takes place. This is especially useful in the following example. Take two integers that form a 64-bit data segment of an incoming bit-stream. The two integers are the first 64 bits of

the incoming bit-stream. After one cycle of the decoding process, 20 bits are decoded. The next cycle of the decoding process needs a variable that contains at least 24 fresh, coded bits. This variable can be made in the following way:

```
new = ((int_1 << pos) | ((int_2 >> 1) >> (31 - pos)));
```

In this case "pos" is 20. Therefore the bits of int_1 are shifted 20 positions to the left. This results in the first bit being a fresh new bit. Int_1 now contains 12 fresh bits, which needs to be extended to 32. The next 20 fresh bits are located at the first 20 positions of int_2. Int_2 needs to be shifted over (32 - 20 =) 12 positions to the right to make a concatenation between the two integers possible through an or-operation. This shift is done in the special way as explained. The or-operation gives the new variable the needed value.

If-else conditional shifting

To be sure about the merged result of a shift over 0 to 31 positions, an if-else conditional shift is possible. The integer is shifted over 0 to 31 positions, so the result is always defined on any 32-bit processor. The position value ranges from zero to the number of bits in the incoming bit-stream. This value grows larger than 31. When it does so, some bits are ignored. The seven least significant bits are taken to specify the shift amount. All other bits specified in the shifting parameter are not taken into account. This results in a kind of modulo operation, which is not executed explicitly. The explicit modulo 32 operation used as the 'if' statement can result in zero as an outcome. This result is the unique case that has to be treated separately. Here the if-else construction is chosen to solve this problem.

This results in the following source code.

```
if ((position % 32) == 0){
    new = int_1;
} else{
    new = ((int_1 << pos | (int_2 >> (32 - pos)));
}
```

This results in a branch in the source code that has a negative effect on the performance. On the other hand, the compiler and scheduler may optimize the operation. A type of optimization in this context is a guarded operation. The 'if' statement forms the guard bit of the operation that decides whether the operation is executed.

Question mark conditional shifting

Another operation that assures the merged result of a shift over 0 to 31 positions is a question mark conditional shift. This operation is added since the effect of either of both conditional shifts on the compiler and scheduler are not predictable.

Therefore both options are compared and explained. A question mark shift is written in one line of source code as shown below.

```
new = ((position % 32) == 0)?
      int_1 :((int_1 << pos) | (int_2 >> (32 - pos)));
```

In this line the statement to be checked and both instructions are specified. This is the only viewable difference between the if-else shift and the question mark shift. The effect on the source code is measured by comparing the number of instruction cycles needed when the same incoming bit-stream is decoded.

Removal of "goto" statement

The original code contains a "goto" statement. This instruction is not the most favourable instruction in the source code of a C-program. The "goto" statement does not result in a clear readable code [5]. Therefore the usage of this statement should be avoided. In the source code written for the TM64 core, the "goto" statement is used to emphasize loops in the source code. This results in better performance since this statement triggers the TM64 compiler to discover a loop. When optimizing the source code, the "goto" statement appears to be useless. Using the "goto" statement does not trigger the TM32 compiler to discover a loop. Therefore this statement is replaced by an equal function as in the source code written for the TM64 core. Here it was used as the exit from the "infinite" loop. This is achieved by giving a break instruction instead.

4.5 Removal of data update branches

The source code written for the TM64 core checks after each decoding step the number of available bits left for decoding. If this number is lower than 24, fresh bits from the bit-stream are fetched. The number 24 comes from the largest possible number of bits needed for a decoding step specific to MPEG-2 decoding. Checking after each decoding step the number of available bits left takes at least three cycles for the execution of the branch. If an update needs to be done afterwards, more cycles are used. The optimized source code proposes rewriting the refreshment. Checking before updating involves a branch. This branch needs to be avoided. As discussed before, the "infinite" loop needs to contain as little branches as possible, because introducing a branch instruction in the "infinite" loop limits the options for loop-unrolling and costs cycles for evaluating the branch. The refreshment policy for the TM64 core with branches is stripped down to a repeatable refreshment policy. Repeatable means that the refreshment policy is put in the "infinite" loop. Each time the loop is executed fresh data is loaded. This data is shifted according to the code length found in the execution of the previous loop. The source code for the refreshment is as follows:

```
if ( acc_code_L >= WIDTH_OF_CPU32_REGISTERS) {
    MSDW_VLC_string_chunk = LSDW_VLC_string_chunk;
```

```
LSDW_VLC_string_chunk = *VLC_string_current_pointer++;  
acc_code_L -= WIDTH_OF_CPU32_REGISTERS;  
}
```

Applying the proposed new refreshment policy results in the following source code:

```
MSDW_VLC_string_chunk = VLC_string[acc_code_L >> 5];  
LSDW_VLC_string_chunk = VLC_string[(acc_code_L >> 5) + 1];
```

This approach does not appear to be more performing than the previous approach. Although a branch condition on loading new data is removed, the cycles for the branch instruction are spent in the unconditional load. Loading data every time the loop executes takes the cycles that were removed by removing the branch instruction. Moreover, since the data is meant to be used directly afterwards, very little code can be scheduled after the load. The emerged source code for the new refreshment policy allows for further optimization. This situation is discussed in the next section.

4.6 Pre-loading data

Loading data from a register takes three cycles in which very little other instructions can be scheduled because the loaded data is used directly. Executing such load operation every time the "infinite" loop runs is quite costly. Especially since this load operation was chosen in favour of a conditional load operation. When conditions for a conditional load are set carefully, the average number of cycles needed per execution of the loop can get below three cycles. This is less than the three cycles needed for the load executed each time the loop is executed. However, this situation can be optimized, which has already been done in the conditional load case. When a load operation is done, but the retrieved data is not used immediately, the scheduler can schedule other operations in parallel with this load. This saves cycles that were previously used for the conditional load, or for waiting until the load operation had finished. In this case a pre-load is done. Data to be used in the near future is loaded, but not used immediately. In parallel with this operation, the already pre-loaded data is shifted and copied into another variable. This uses more resources from the TM32 core, but forms no constraint on the possible performance of this implementation.

4.7 Stand-alone Entropy Decoding on TM32 Conclusion

Since Entropy Decoding is an essentially sequential task and lies on the critical path of MPEG-2 Decoding it is worth optimizing this task. The TM32 benefits from Instruction Level Parallelism (ILP); this is exploited intensively to give a highly performing Entropy Decoder. The steps taken to get this result are:

- Porting the available TM64 Entropy Decoding source code to TM32.
- Re-ordering and compressing the data in the Lookup Table; this reduces load and branch operations.
- Analyzing the incoming bit-stream to determine the order of branches in the Entropy Decoding source code.
- Identifying unique cases in the loop that is executed frequently to reduce branches.
- Removal of branch operations for loading new data.
- Pre-loading data

These six optimization steps transform the Entropy Decoding source code completely. This results in a significant decrease in the number of cycles needed for the Entropy Decoding step. In addition to this the slot occupancy of VLIW slots increases. The slot occupancy gives a measure of the level of optimization. This indicates how intensive the source code is optimized and how much room there is for further optimization steps. The intensiveness is expected to be high whereas the room for further optimization steps is expected to be little.

Entropy Decoder Integration in Multi-threaded MPEG-2 Decoder

5

The performance of the suggested Entropy Decoder (chapter 4) requires integration on a performing multimedia-processor. A processor in this class is SpaceCAKE. First the Entropy Decoding blocks in the MPEG-2 Decoder written for SpaceCAKE need to be identified. At those points the old source code should be replaced. With the replaced source code a new data management is needed to switch between the MPEG-2 Decoders own character organization and the new integer organization. Because of a new organization in processor threads the initializations need to be moved to such a place that these are executed once per processor. This gives a better maintainable source code, as does the merging of the Entropy Decoding blocks in only two separate functions.

Section 5.1 deals with the identification of Entropy Decoder Blocks. The patches needed to switch between data management styles are explained in section 5.2. Section 5.3 explains the movement of initializations. For better maintainability, the Entropy Decoding blocks are merged into two separate functions as explained in section 5.4. Section 5.5 discusses the obstacles encountered during these processes.

5.1 Identification of Entropy Decoder blocks

Entropy Decoding consists of a variety of block types that is decoded. This block type can be read from a header parameter when decoding a bit-stream. Possible block types are:

- Non-Intra blocks
- Intra blocks starting with a Chrominance coefficient followed by
 - Coded symbols to be looked up in table B14
 - Coded symbols to be looked up in table B15
- Intra blocks starting with a Luminance coefficient followed by
 - Coded symbols to be looked up in table B14
 - Coded symbols to be looked up in table B15

Each of these block types requires a different method of decoding, resulting in at most five separate parts of source code.

A source code for MPEG-2 decoding on SpaceCAKE is available. This source is based on the standard Berkeley MPEG-2 decoder programmed in C [10] [4]. This decoder

decodes each block type in a separate function. The functions used decode the following blocks:

- A Non-Intra block
- A Chrominance block
- A Luminance block
- A B14 coded block
- A B15 coded block

The available source code written for the TM32 core is divided in five functions (see above). Although this source code is also based upon the Berkeley MPEG-2 decoder, the functions are not interchangeable. The TM32 source code is much more developed and optimized which makes it a completely different code from its SpaceCAKE equivalent. Enabling usage of the TM32 source code requires replacing the source code at a higher level. This involves a patch that transfers the character organized data management to the optimized data management used in the TM32 source code. This is explained in the next section.

5.2 Adapting data management

The data management in the standard Berkeley MPEG-2 decoder is implemented in a costly way. A lot of performance is wasted in executing function calls and character orientation.

The first performance waste, calling a function, appears when new data is read and when the data buffer is updated. Reading data is done by a function named "show bits". Updating the data buffer is done by a function named "flush buffer". Executing a function call results in a cost of three cycles to get from one function to the next. Executing such operation in the "infinite loop" of the Entropy Decoder gives opportunities for a large gain in performance when these calls are eliminated.

The second performance waste, character orientation, appears when the data buffer is updated. The buffer is updated character wise on a 32-bit processor. Filling one integer character wise needs four loads and some overhead to get the data in one integer. A load takes three cycles, whether it is an integer or a byte. Replacing the character load by an integer load therefore results in a gain in performance.

When adapting the data management of the MPEG-2 decoder, the whole management should be rewritten. This has a large impact on the MPEG-2 decoding source code since function calls on data management functions are done throughout the whole program. Each usage of a function call requires a custom fit rewritten source code in a general style to get around such call. The custom made code is needed to make each part of rewritten code fully testable. The final goal is merging the local solutions to one global solution. This gives opportunities to skip a local work around that patches from and into the optimized Entropy Decoder.

As mentioned, rewriting all appearances of data management function calls is a time consuming job. Therefore rewriting is focused on functions involving Entropy Decoding. The approach taken is adapting the Berkeley data management to the optimized data management. This is done by copying the expected amount of data needed into a locally used data-stream. This patch results in some overhead. On the other hand, working with this patch gives better performance in the Entropy Decoder function. It is expected that this result outweighs the overhead of the patches needed. It says patches because not only one patch is needed when going to the function that uses the optimized data management, but also when it returns from it.

A simplified character organized update patch is written to be executed when returning from the optimized data management. This patch is needed to enable the usage of the character organized data management. The patch consists of a jump in the original data-stream. This jump is based on the total number of bits decoded when the optimized data management is active. This assures the ability to follow the incoming bit-stream correctly. This jump cq patch consists of the following steps:

- A rough jump based on the number of integers decoded.
 - This jump sets the character pointer to the buffer used in the character organized data management.
- Added to this a detailed jump based on the number of bits left to be jumped over after the rough jump
 - This jump makes sure the correct data is loaded into the locally used character organized data buffer.
- Finally the number of new bits in the locally used buffer is set.

The condition of the usage of this patch is that it can only take place when no switch between buffers is needed. The buffer mentioned is the large buffer containing the original data. When the decoder comes to the end of a buffer a special set of instructions is executed to get to a new buffer. This code is not rewritten. Instead of rewriting this code, a branch instruction is written in source code. This makes sure that when a buffer switch is very likely to be needed, no patch and no optimized entropy decoder is run. When the original data buffer from the Berkeley MPEG-2 decoder is large, the effect of the remark made is only measurable when a very detailed look is taken at the results and performance. The loss in performance is negligible compared to the extra possible gain when a source code is written that does not need this branch. Writing this code is not feasible because of the large effects this has on the basis of the Berkeley MPEG-2 decoder. Therefore this is skipped and turned into a branch that maintains character organization when a buffer switch is at hand.

5.3 Inserting initializations and Lookup Tables

The initialization of the Lookup Tables for Entropy Decoding in the optimized Entropy Decoder takes place within the function. Taking this approach in the SpaceCAKE source

code as well results in many more initializations than needed. This emerges from the fact that the Berkeley MPEG-2 decoder calls the Entropy Decoding function separately for each macroblock. However each processor only needs a single Lookup Table initialization. Therefore executing the initialization steps is reduced from each time the Entropy Decoding function is called, to once for each processor in SpaceCAKE. This results in placing the Lookup Table initialization among other initializations that are executed for each processor at the start of the MPEG-2 decoder source code.

5.4 Merging Entropy Decoding into two functions

At this point, the Entropy Decoder is spread over many functions. As explained in section 5.1, the five possible ways of decoding each have their own function. Actually these five possible ways are put into two categories. There is the possibility for a Non-Intra block or an Intra block when decoding an incoming bit-stream. All other possibilities are subsets of the Intra block option. When writing an Entropy Decoder with this view, the number of functions is reduced and the maintainability of the source code profits from this. In addition, the number of function calls is reduced, resulting in a gain in performance. However, rewriting any source code to become multi-functional affects the performance of it. Choosing identical operations that initialize block types within the merged function minimizes this effect on the performance. These operations choose the Lookup Table and choose the number of bits to be read from the incoming bit-stream. The instructions needed for this are already present in other parts of the source code. Here they are followed by a function call that jumped into an Entropy Decoding routine for a block type. This last part is eliminated by the discussed merge. The final result is a larger function with more operations and branches inside. There are no new branches introduced in the MPEG-2 Decoding process and the number of function calls is reduced. This combination has a positive effect on the performance. In addition the opportunity increases to schedule instructions in parallel. Therefore this step not only results in a more readable code, but it becomes also better performing.

5.5 Obstacles encountered during development phase

When all Lookup tables are initialized properly and the rest of the source code is in place, the Entropy Decoder follows the incoming bit-stream errorless. But this does not ensure a fully correct output of the Entropy Decoder. The image needs to be checked for errors. When a value in a block is slightly wrongly calculated, this is not seen immediately in the image block itself. However, the error starts to accumulate shows up in the blocks following the erroneous one. This makes these errors mainly occur in the latter part of an image line. When these lines turn black or any other colour, the calculation of a displayed image block at the beginning of the slice is erroneous. This is the problem to be tackled in the Entropy Decoding source code written for the TM32 core. This source code shows the optimization possibilities for a stand-alone Entropy Decoder. It performs all steps that need to be taken, i.e. all run-level pairs are decoded correctly from the incoming bit-stream. However, no specific care is taken of creating the correct

output picture. The steps needed for these calculations have been taken into account. Nevertheless, the correct functionality needs to be tackled at this point.

When integrating the Entropy Decoder in the fully functional MPEG-2 decoder written for SpaceCAKE it generates the final picture from the video-stream. The picture made with the optimized source code shows that some major adjustments are needed. These adjustments involve rewriting the source code that creates the image blocks after Entropy Decoding. This code was not focused on when writing the optimized Entropy Decoder. It only needed to be functionally correct to prove the performance of other optimization steps. Therefore the image construction in the optimized Entropy Decoder is adapted to conform to the Berkeley MPEG-2 Decoder results. This makes a performance comparison between the two fair. No side effects are taking place that affect the performance. The only performance gain is found in the new Entropy Decoding approach.

The SpaceCAKE source code that creates the image blocks narrowly fits the original optimized source code on all but one point. A small change in variable names and calculation steps gives major improvements in the readability of the source code. This results in an almost correct image. The luminance and chrominance values are not processed correctly yet. A dummy value occupies their place in the image block. This dummy value is to be replaced. At first a complete image block is created according to the B14 or B15 coded symbols. The Luminance or Chrominance values are calculated and stored separately. When the image block is filled, the first value is replaced by the needed Luminance or Chrominance value. The result is conform to the result from the Berkeley MPEG-2 Decoder.

5.6 Entropy Decoder Integration Conclusion

Integrating the optimized Entropy Decoder in a MPEG-2 Decoder involves a set of patches. This is due to the difference in data management organization. In the first place the Entropy Decoder blocks need to be identified. These are placed in various locations throughout the source code, which is a different approach from the one taken in the optimized Entropy Decoder. Therefore these blocks all are merged into two Entropy Decoding functions. This reduces the amount of function calls and increases the maintainability of the source code. These steps have resulted in a fully functioning MPEG-2 Decoder. This decoder needs to be checked against the Berkeley MPEG-2 Decoder, because all MPEG-2 Decoders need to conform with this decoder. Due to this conformance check some errors in the resulting output have been repaired.

6

Experimental results

This chapter shows the experimental results in a chronological way. No experimental results about the TM64 source code are included. These results have been reported in several papers which are referred to when applicable.

Section 6.1 deals with the experimental framework in which the experiments have been done. The first results are about the Entropy Decoder written for TriMedia/CPU32. These results are presented in section 6.2. Section 6.3 shows the results for a full MPEG-2 decoder written for SpaceCAKE. The optimized Entropy Decoder is integrated in this MPEG-2 decoder.

6.1 Experimental framework

The experiments of the Entropy Decoder written for the TM32 core are executed on a machine level simulator in software. The results of these simulations are visualized using the Proview tool. Proview is a software tool that is only available within the Philips Research Laboratories domain. The tool visualizes in three colours the degree in slot occupancy of VLIW issue slots per function or group of functions. In addition the exact numbers belonging to the selected data is printed below the visualization. The experiments on the Entropy Decoder written for SpaceCAKE are executed on a machine level simulator in software as well. The essential simulation data is returned when exiting from the simulator. In addition the final picture from the video-stream is depicted as well. This picture shows the quality of the decoding process from which the adjustments needed are deducted.

6.2 Stand-alone Entropy Decoder performance

The stand-alone Entropy Decoder is a part from the MPEG-2 Decoding process that artificially is extracted and optimized for the TM32 core. This decoder is written to show the performance of Entropy Decoding as an operation on its own. It is aimed to show the advantages of a multimedia processor with a VLIW core. The VLIW principle is explained in section 3.1, which also explains other TriMedia and SpaceCAKE properties.

As explained in section 4.3 three different cases are considered. Each is considered throughout all optimization steps of the source code. The three different cases are:

- Smart non-conditional shifting
- If-else conditional shifting
- Question mark conditional shifting

The optimization steps are based on the following principles:

- A 64-bit result coming from the Lookup Table; two integers containing non-packed data
- A 32-bit result coming from the Lookup Table; one integer containing packed data
- Pre-loading data from the incoming bit-stream; loading data into the register file

These three by three variants result in nine different cases that are considered. Each of the optimization steps is shown for all cases in a separate chart. The optimization steps give an improvement in performance. Therefore no straight comparison between optimization steps is made. The graphs give an insight in two major characteristics involving performance. At first the number of cycles needed to decode a coefficient is shown. A lower number of cycles means a better performance. Secondly the number of issued instructions per cycle is shown. A larger number of issued instructions shows a better usage of available resources. The maximum number of issued instructions per cycle is five. When this number is approached a very little performance gain can be achieved by exploiting more parallelism. The only possibility left for a larger gain in performance is removing trivial instructions. Since each instruction is considered carefully, this option can be ruled out as well.

The graphs are based on six well-known input test-streams for MPEG-2 decoding. These streams are called:

- Bat_327_334
- Popplen
- Queen_2248_30
- Sarnoff2
- Tennis
- Tilcheer

These streams represent a variety of MPEG-2 streams with different properties concerning block types and number of non zero coefficients. Possible block types are as explained in section 5.1:

- Non-Intra blocks
- Intra blocks starting with a Chrominance coefficient followed by
 - Coded symbols to be looked up in table B14
 - Coded symbols to be looked up in table B15
- Intra blocks starting with a Luminance coefficient followed by
 - Coded symbols to be looked up in table B14

- Coded symbols to be looked up in table B15

The number of non-zero coefficients is specified per block. In this case it ranges from three in Queen_2248_30 to eight in Popplen. A non-zero coefficient has a run-level pair. Run specifies the number of zeros of preceding the level coefficient (see section 2.1).

Each graph shows the results for the different shifting methods (section 4.3). In addition, the regular and grafted options appear. The regular option means no optimization flag has been set when compiling and scheduling. The grafted option means that the default optimization level O3 is chosen (section 3.1).

For readability and comparison reasons, all shading used in the graphs is kept the same throughout this section.

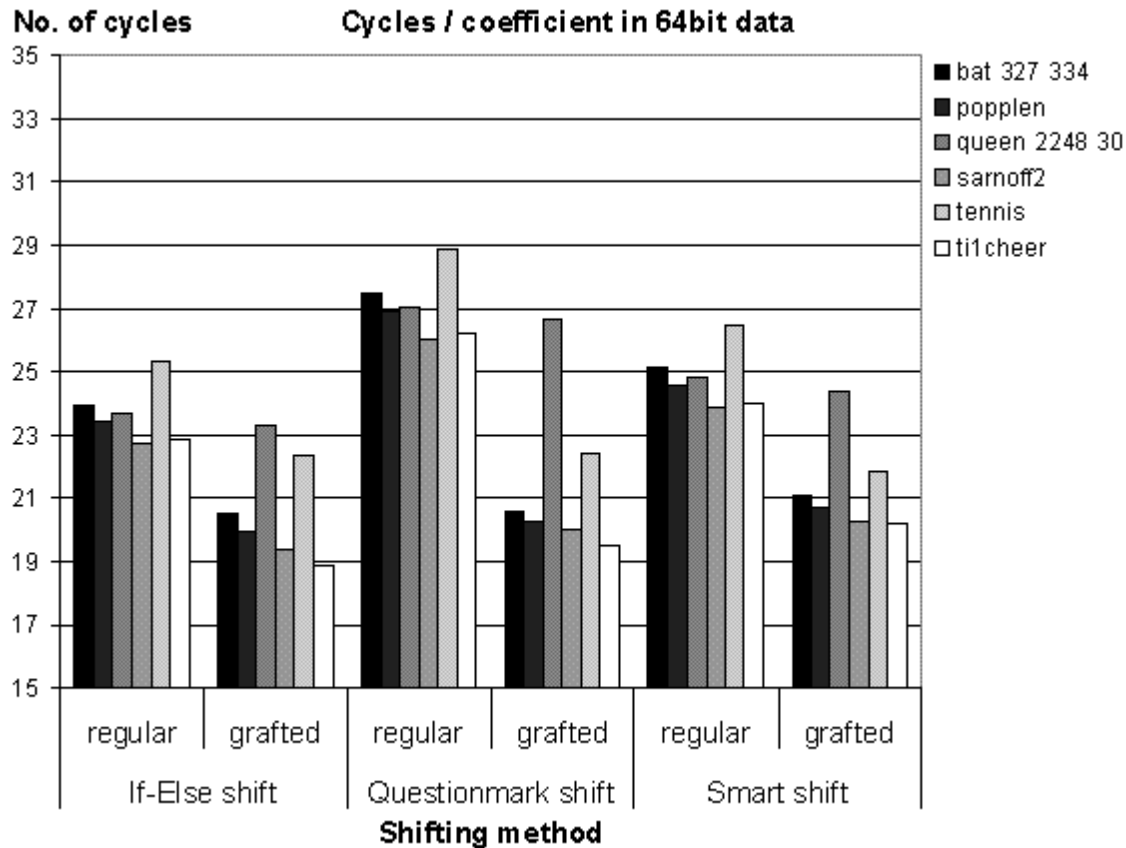


Figure 6.1: Entropy Decoding in cycles per coefficient with 64-bit data from the Lookup Table

The first experimental results come from the rebuilt source code written for the TM64 core. No optimization steps have been carried out yet. Grafting (section 3.1) results in some optimization since about three to seven cycles less are needed to decode a coefficient. This results in an optimization of 13% to 26% less cycles compared to the non grafted version. The three shifting methods in Figure 6-1 show very different results. The If-Else shift takes the least cycles needed in a regular or grafted case. The Question-mark shift is optimized best by grafting. The Smart shift appears to be average in results compared to the If-Else and Question mark shift options. It shows no special properties at this point. Another remarkable result of the Smart shift is the number of cycles needed for one coefficient in the Queen and Tennis case. This is about two cycles more than the other cases. This is due to the type of coding used for these streams. These two streams are similar to a High-Definition stream, which uses more cycles to decode a coefficient.

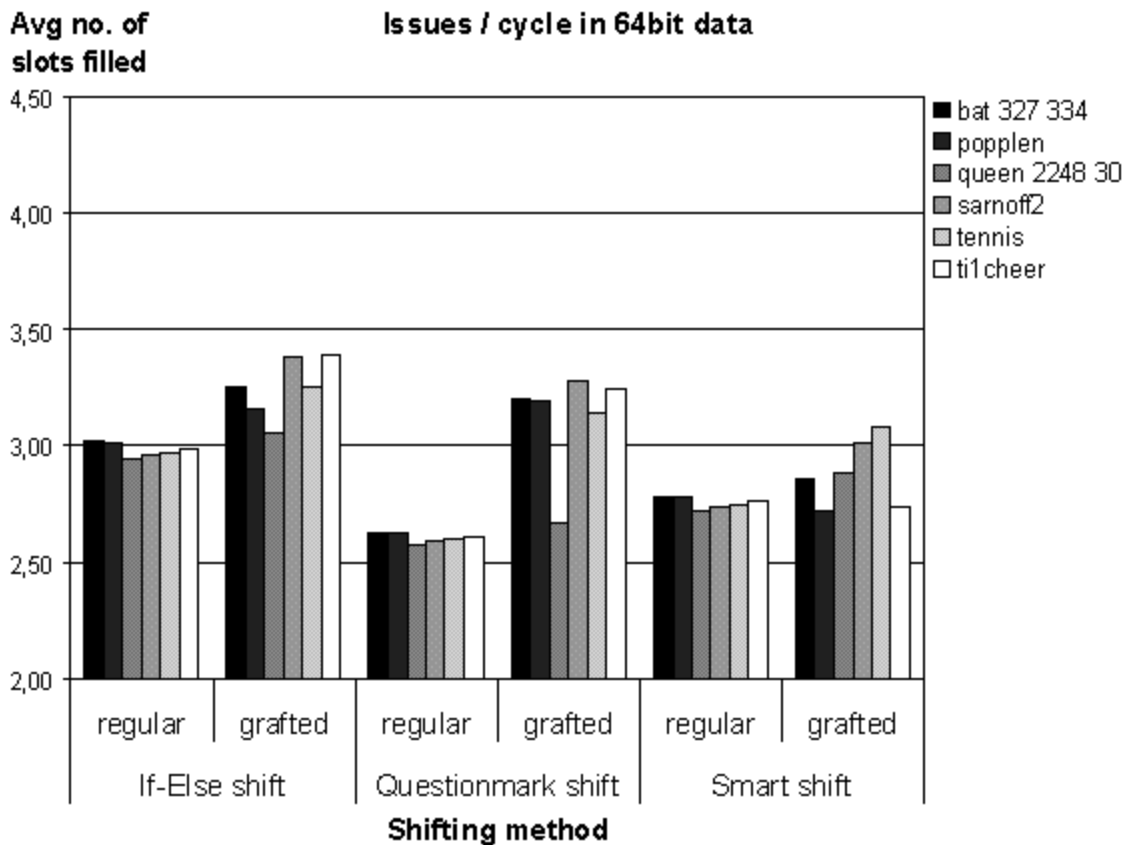


Figure 6.2: Entropy Decoding in issues per cycle with 64-bit data from the Lookup Table

Grafting results in a higher slot occupancy of a VLIW slot, e.g. a 5% to 45% higher slot occupancy (Figure 6-2). The smart shift has little benefit from grafting, since this shift option is hardly optimized for all input bit-streams. The other two shift options show the same optimization through grafting as shown in the coefficients per cycle graph, although this does not come forward very clearly. The performance of If-Else shifting gives the best results for ILP. The difference in slot occupancy of VLIW slots is the largest in the Question mark shifting case showing the largest gain in ILP. This graph also shows the special case of the Queen and Tennis test-stream. Using grafting in these two cases has less effect on the slot occupancy of VLIW slots. Especially the Question mark shift in the Queen case shows this in an extreme way among the other results. Here an improvement of only 5% is achieved compared to the 40% to 45% in the other streams.

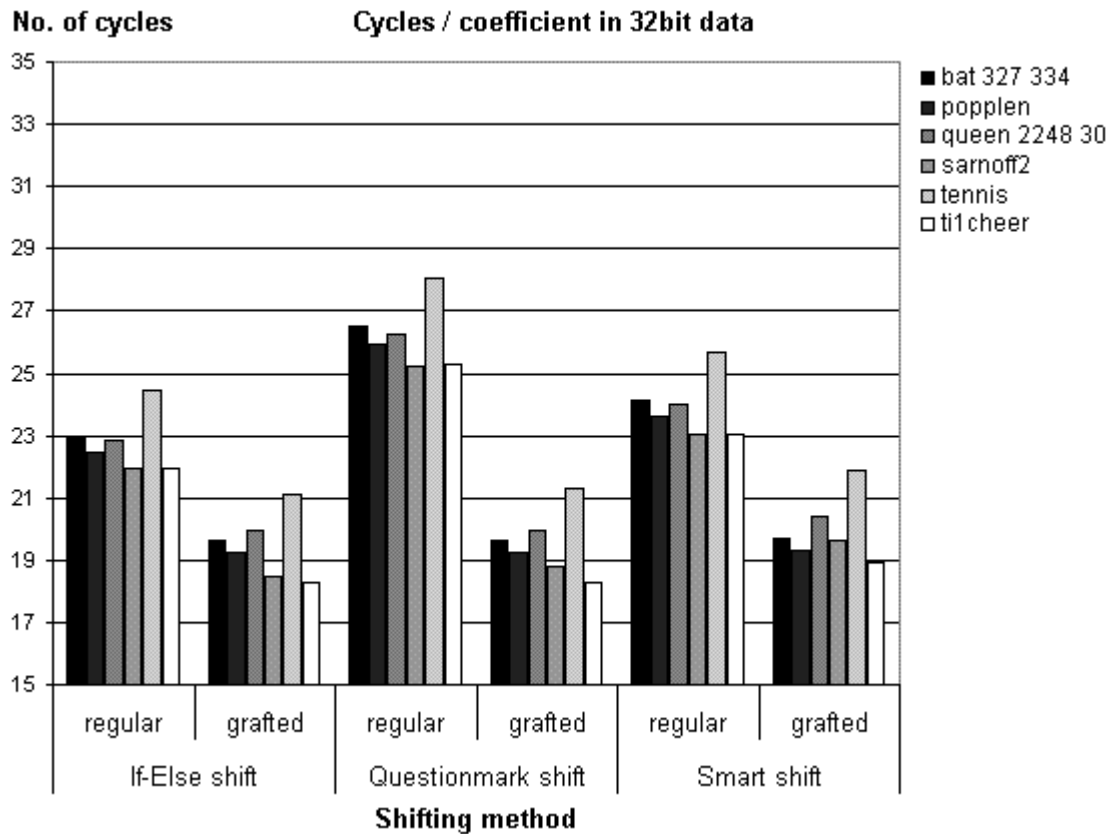


Figure 6.3: Entropy Decoding in cycles per coefficient with 32-bit data from the Lookup Table

Figure 6-3 and Figure 6-4 show the difference that comes from reducing the amount of data coming from the Lookup Table. At first this data was put in two 32-bit integers. In this case one 32-bit integer is used. This optimization step shows two improvement advantages compared to the previous situation. The first improvement is a global one. All numbers representing the number of cycles per coefficient are reduced with at least one cycle per coefficient. This is due to fewer operations needed for data retrieval from the Lookup Table. The second improvement is shown by the result of grafting. Grafting works in this case better for the Queen and Tennis bit-stream, although these bit-streams still appear to be a little awkward. Compared to the previous results the graph shows a more regular result. Again an identical judgement on performance when comparing shifting methods. The If-Else shift performs best, whereas the Question-mark shift optimizes best. Smart shifting again is the least favourable option.

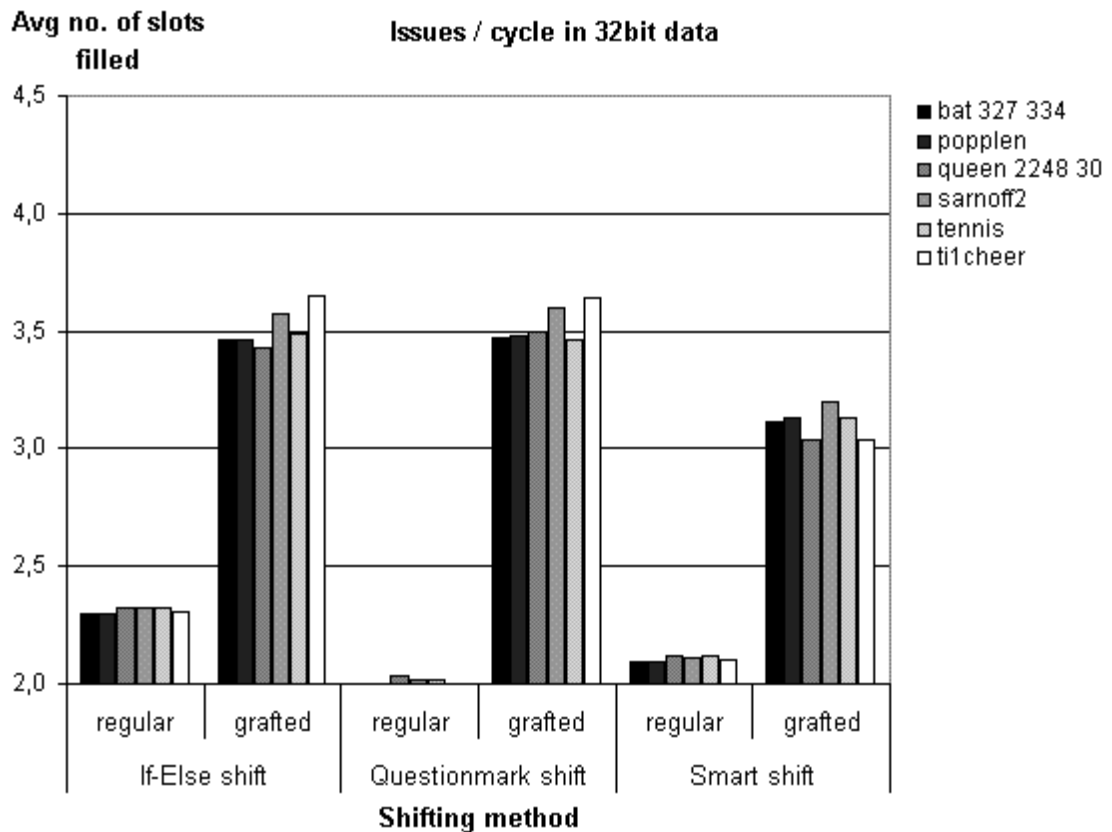


Figure 6.4: Entropy Decoding in issues per cycle with 32-bit data from the Lookup Table

Here a clear result on the ability to optimize the source code is shown. The source code consists of fewer instructions that are organized in a similar way as before. Fewer issue slots in a VLIW slot are filled in, resulting in a low slot occupancy for the regular source code. When grafting is applied, the result shows a large difference in performance. The VLIW slot occupancy increases (Figure 6-4) showing a difference of more than 50%. This shows that the source code is able to exploit more ILP, is positive for performance results on VLIW cores. In this graph no difference is seen in the performance of the special streams. Both of these, the Queen and Tennis test-stream, which have equal results compared to the other streams. This is different for the shifting methods. These do not appear to have changed very much. Their behaviour in this situation is equivalent to their behaviour in the previous situation. Although the Question Mark shift results in a slightly higher slot occupancy of VLIW slots, this difference is negligible.

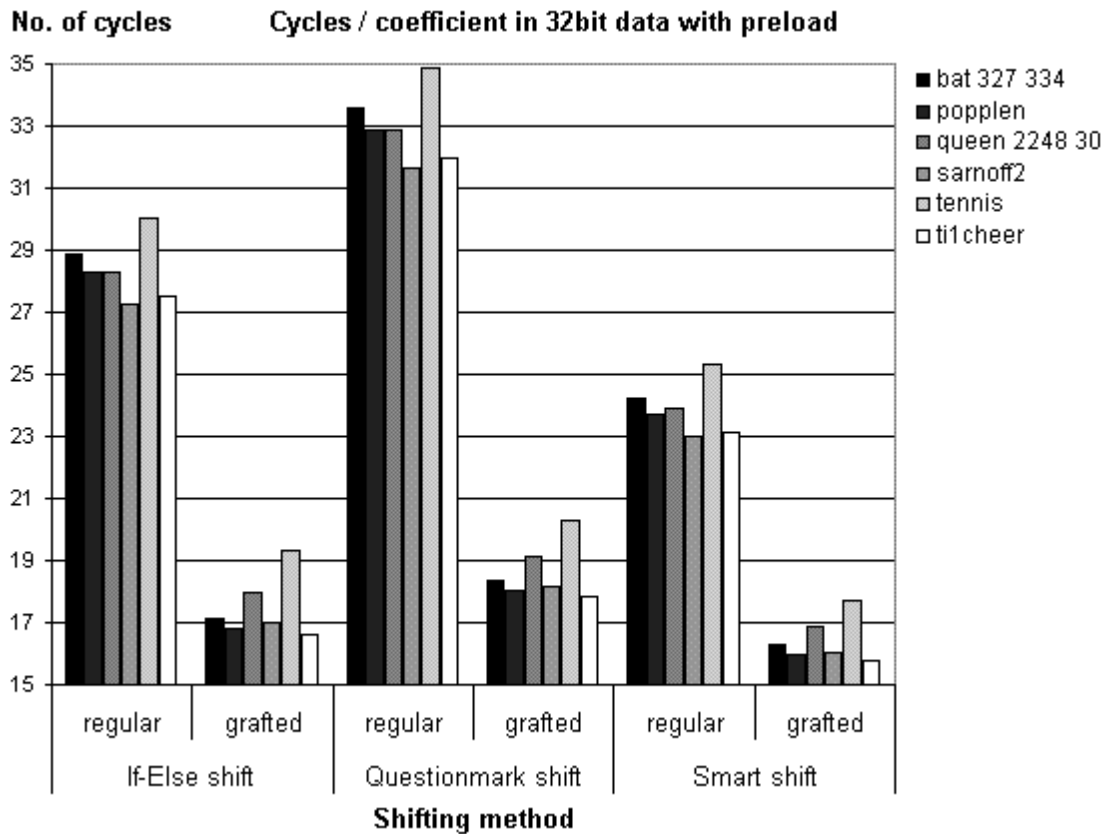


Figure 6.5: Entropy Decoding in cycles per coefficient with pre-loading of data from the Lookup Table

Writing a pre-load method results in a lot more source code, and a growth in cycles appears using the regular source code. This growth is drastically reduced when grafting is used (Figure 6-5) which shows that the ability to parallelize this source code has grown. On the other hand the number of cycles needed to decode a coefficient has decreased. A reduction of about two to three cycles is achieved in this way. The most remarkable result is the performance of the Smart shifting method. This method performs a lot better with pre-loading data, these results were not very promising in previous situations. In addition to this, Question-mark shifting remains to show its large ability to be optimized. However the final result of this optimization is the worst, due to the starting point of the optimization step that uses a lot more cycles. In this situation the awkwardness of the Queen and Tennis input streams returns. On the other hand these streams show equal possibilities for optimization.

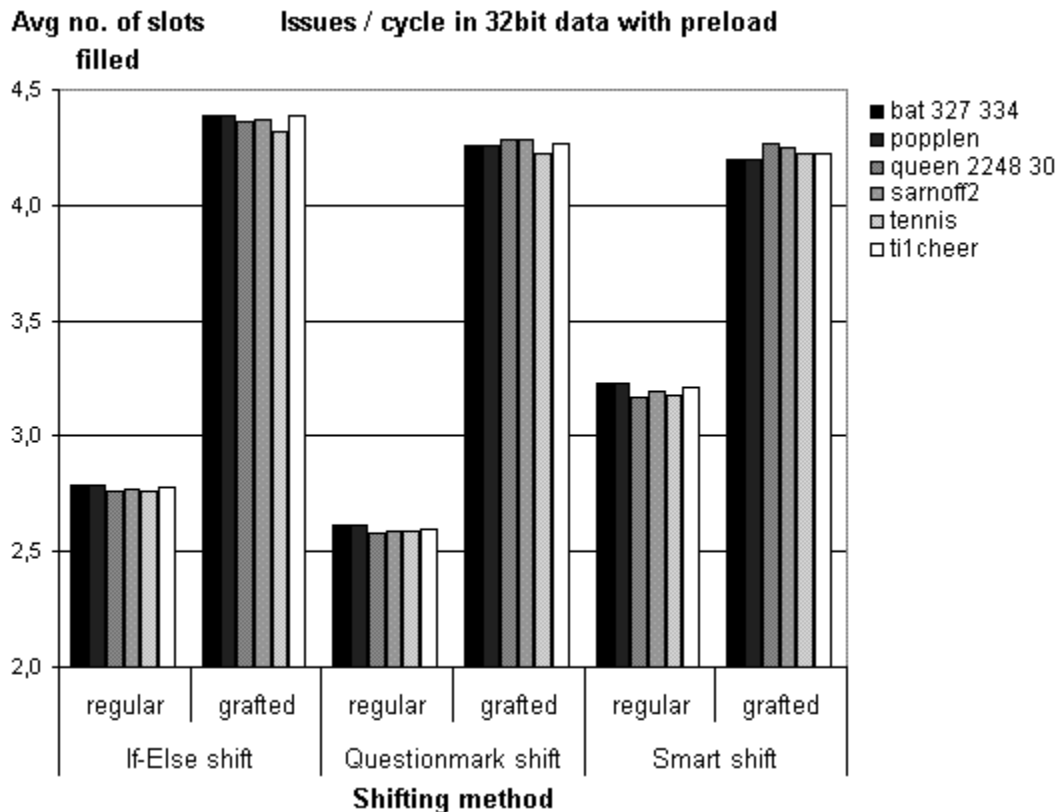


Figure 6.6: Entropy Decoding in issues per cycle with pre-loading of data from the Lookup Table

Using pre-loading is a way to introduce parallelism. Figure 6-6 supports this idea. On the whole, half of the issue slots in a VLIW slot are filled in addition to the previous situation in the regular case. When using grafting, this number goes up to almost one complete extra instruction slot in a VLIW slot. This gives a slot occupancy approaching 90% of a VLIW slot, an exceptionally high degree. It shows that this source code for Entropy Decoding is highly optimized. The 90% slot occupancy is expected to decrease when the stand-alone Entropy Decoder is integrated in a fully functional MPEG-2 decoder since this is an artificial situation. Another remarkable result is the average slot occupancy of VLIW slots. This number is almost identical for each of the five bit-streams in the different situations. This shows that the ILP is independent of the incoming bit-stream for each shifting method. The If-Else shifting method gives the best slot occupancy of VLIW slots. But its performance in number of cycles per coefficient needs a cycle more than Smart shifting method. This last method is the best performing.

6.3 MPEG-2 decoder on SpaceCAKE

The MPEG-2 Decoder written for SpaceCAKE is a fully functional decoder. It returns the final picture of a decoded MPEG-2 video bit-stream. This result does not only give information about the ability to follow this bit-stream. It also shows whether the calculations that form the pictures in a video-stream are executed correctly. After integrating the stand-alone Entropy Decoder, some tests have been run to show how many cycles in which functions are spent. This is displayed for the six largest functions. The test input-streams are two available extremes. These input-streams are extreme in their number of non-zero coefficients in a block. The Queen-stream has three non-zero coefficients per block. The Popplen-stream has eight non-zero coefficients. These results make a judgement possible about what functions dominate the performance results. This judgement enables a decision on minimizing the dominance of less important functions. The experimental results in this section do not show a result in cycles per coefficient for Entropy Decoding. This result does not appear anymore because of the spreading of the Entropy Decoder function over the MPEG-2 decoder. The identification of one Entropy Decoder block is not possible as explained in chapter 5. The experimental results on performance show the total number of cycles needed for MPEG-2 decoding. This number is achieved by taking eighteen well-known MPEG-2 decoder test-streams. These results give a good thorough view on the capabilities of the optimized MPEG-2 decoder. The graphs showing these capabilities in number of cycles and speed up show two curves. These curves enable distinguishing between the optimized and the original version.

Table 6.1: Overview on coefficient density per block for input bit-streams

Stream	no. non-zero coefficients per block	total no. of coefficients
bat_327_334	6,8	439230
popplen	9,1	75072
queen_2248_30	3,5	2091251
sarnoff2	8,0	116971
tennis	7,9	270855
tilcheer	8,6	132498

The effect of the integration of the optimized Entropy Decoder in the multi-threaded MPEG-2 decoder is evaluated using two input bit-streams that are extreme in their properties. One extreme is the queen_2248_30 bit-stream. This bit-stream has the lowest number of non-zero coefficients per block. Furthermore, it has the highest total number of coefficients. The other extreme is the Popplen bit-stream. This bit-stream has the highest number of non-zero coefficients per block. In addition to this the total number of coefficients is the lowest. The queen_2248_30 and Popplen bit-stream are bit-streams that give a representative view on the functions involved in the Entropy Decoding process, because their results show identical features of the integrated Entropy Decoding process and its accompanying functions. Their relations are shown in Figure 6-7 and 6-8.

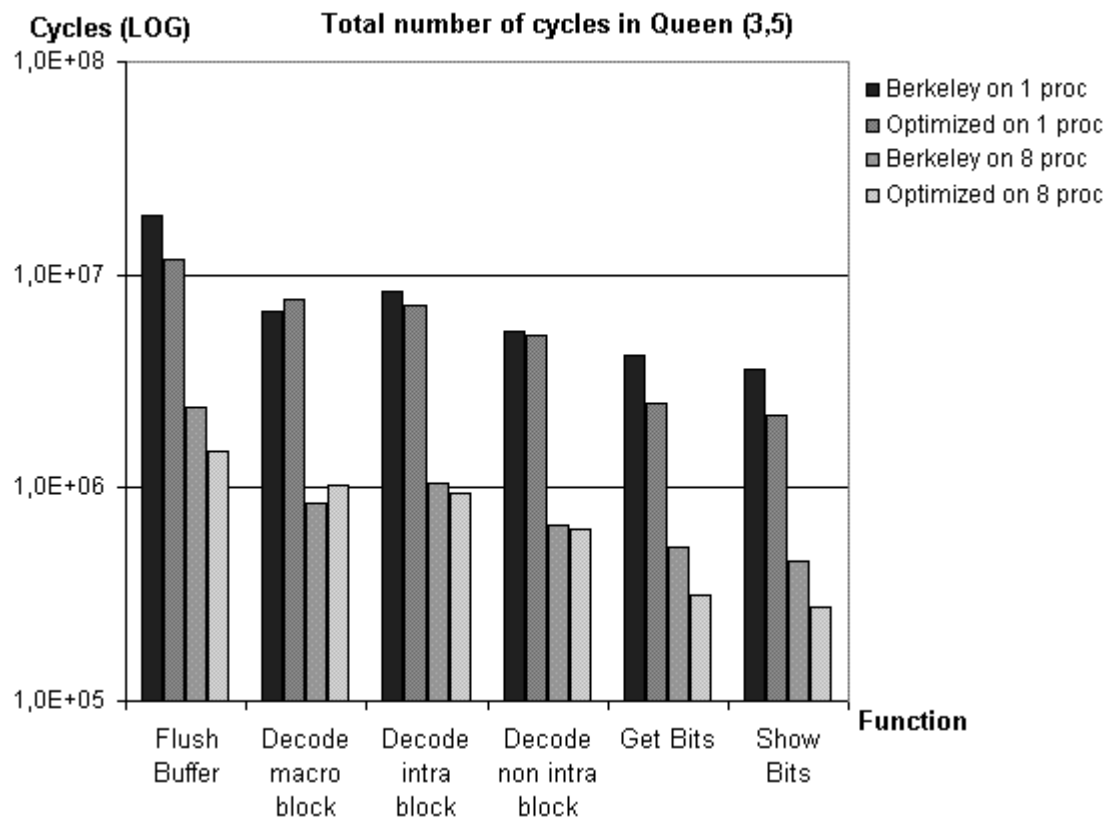


Figure 6.7: Number of cycles in Entropy Decoding and its accompanying functions for queen

The Queen test bit-stream has a minimal average of three and a half non-zero coefficients per Entropy Decoder block (Table 6-1). After decoding three frames of the bit-stream, the results of this process are added up (Figure 6-7). The graph shows the contributions in cycles of the Entropy Decoding process and its accompanying functions in the MPEG-2 decoding process. The accompanying functions, Flush Buffer, Get Bits and Show Bits, show that an equal amount of cycles needs to be executed. These functions do not form an effective performing contribution to the Entropy Decoding process, in that they are data management functions executing their tasks in the background compared to the decoding operations. The approach taken in the optimized version manages to reduce the dominance of data management functions with 40% in cycles, which cannot be eliminated because of the usage of these functions throughout the MPEG-2 decoder (see section 5.2).

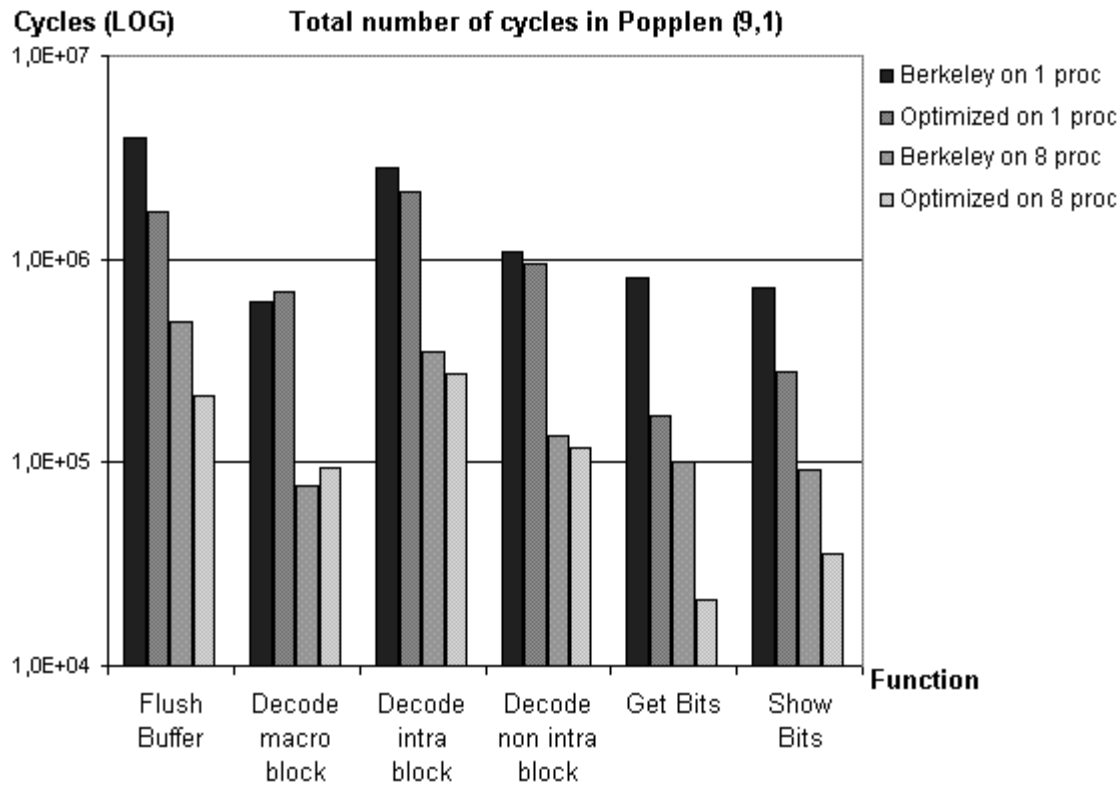


Figure 6.8: Number of cycles in Entropy Decoding and its accompanying functions for Popplen

The results of the Popplen test bit-stream (Figure 6-8) show the same dominance of data management functions as compared to the Queen test bit-stream when decoding three frames. This makes this dominance independent of the number of non-zero coefficients in a test bit-stream. When this number varies from 3,5 in the Queen case to 9,1 in the Popplen case the dominance remains. The only difference is that case the dominance can be reduced with 60% and even 80% for the Get Bits function compared to 40% in the Queen case. Data management for the Entropy Decoding process separately is not shown. The results show that adapting the data management for Entropy Decoding only results in remarkable reduction of instruction cycles needed for the same process. Cycles used to be spent in the data management functions are now cut out and only partially replaced by instructions in the decoding process. Therefore the gain in performance of Entropy Decoding can partially be seen from the parts it consists of.

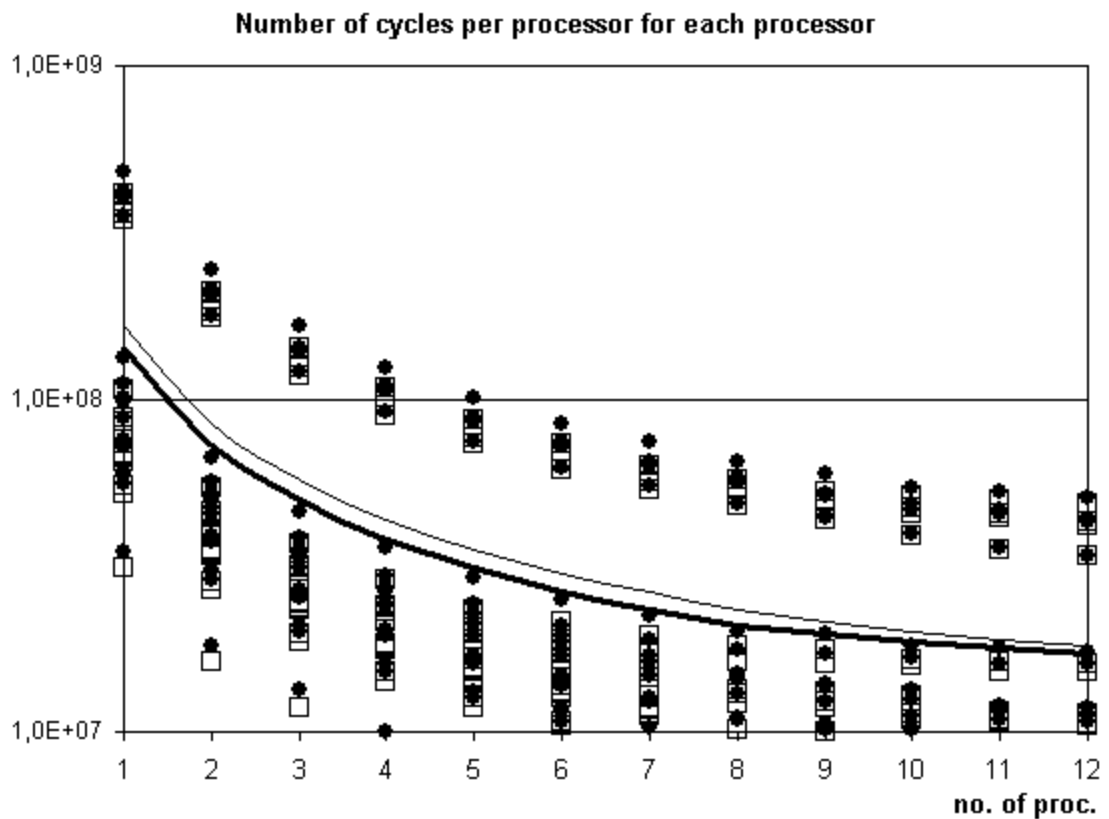


Figure 6.9: Average number of cycles needed for MPEG-2 decoding depending on the number of processors

A process executed in parallel always shows a logarithmic decrease in number of cycles when the number of processors is increased. Added to this a minimal number of cycles is always needed to execute parts of the process that cannot be done in parallel. This is shown in the decrease in cycles of the MPEG-2 decoding process logarithmically whereas it asymptotically approaches a constant value (Figure 6-9). The dots in the graph show the results of the Berkeley decoder. These results have an average shown by the small line. The blocks show the results of the Optimized decoder. These results have an average shown in the thick line. Both lines and data points show the same behaviour for an increasing number of processors. This shows that the optimization step maintains the same properties for optimization.

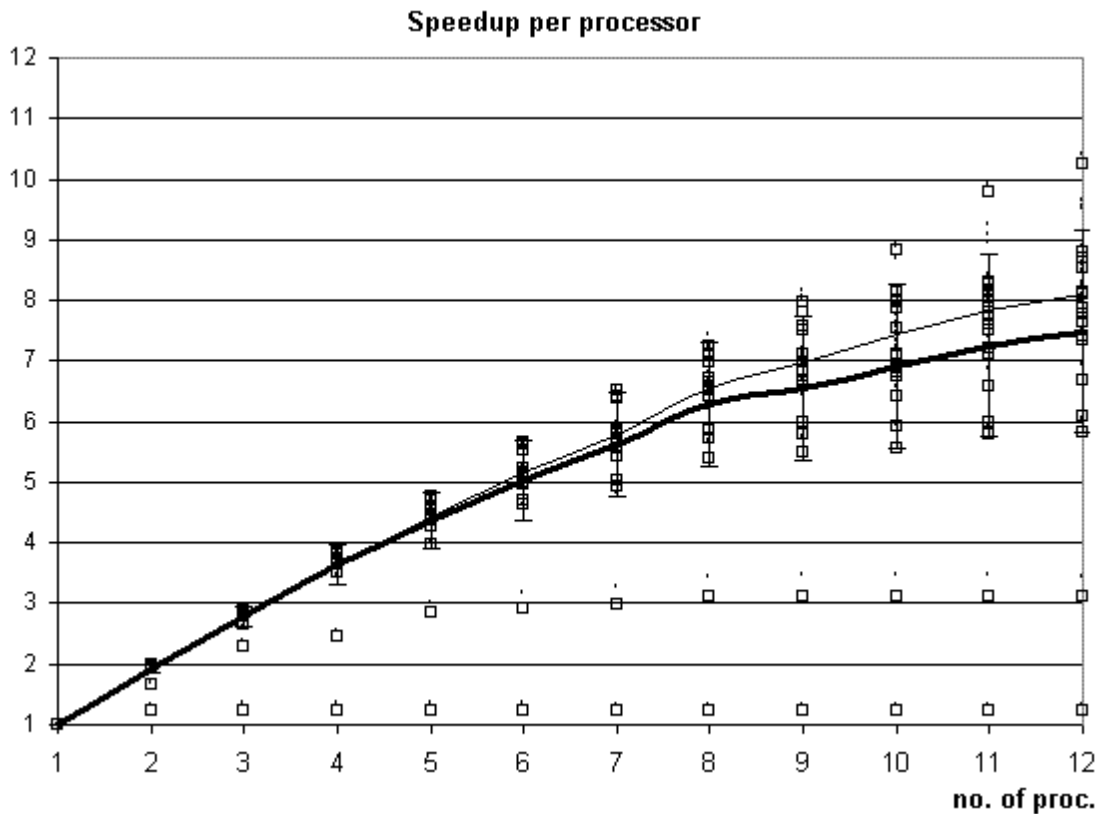


Figure 6.10: Speedup of MPEG-2 decoding per added processor

Ideally adding a processor to a system of multiple processors, gives a speedup of the process, which can be seen as linear. Only this does not take overhead into account. This overhead involves mainly data management functions. These functions form a bottleneck in the optimization of the Berkeley MPEG-2 Decoder. Due to overhead a straight line is shown from one to eight processors (Figure 6-10). This is the part where the speedup of the process can be seen as linear. When more than eight processors are allocated in the system, the speedup per processor decreases. Figure 6-10 shows the same organization of data as the previous graph. The error bars belong to the thick line. The graph can be split into three parts. In the first part, 1 to 4 processors, the lines coincide with each other. In the second part, 4 to 8 processors, the optimized decoder closely follows the speedup achieved for the Berkeley decoder. In the last part the optimized process differs from the Berkeley decoder. The speedup is limited because the overhead starts to dominate the performance of the decoder.

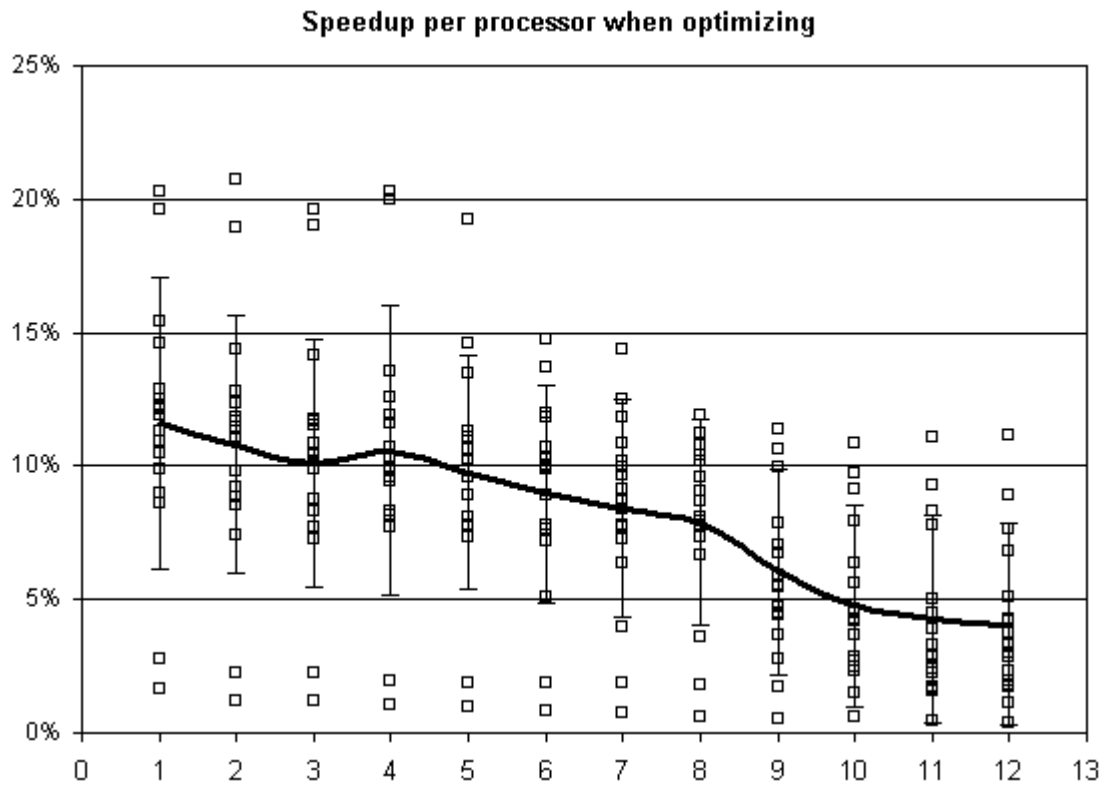


Figure 6.11: Speedup of optimized MPEG-2 decoding per added processor

The difference in performance between the Berkeley Decoder and the optimized decoder is also analyzed using percentages of the originally used number of cycles. These percentages show the gain in performance achieved by optimizing the Berkeley decoder. The line in Figure 6-11 shows the average improvement in performance. The variation shown by the error bars is very large. This variation is caused by the difference in properties of the data. Test bit-streams with a low number of non-zero coefficients per block speed up by a small amount. This difference was already found in the comparison of major functions in cycles. A remarkable result is seen at situations where a multiple of four processors is used. These structures apparently give a better structure for dividing data and operations over the processors.

6.4 Experimental results Conclusion

The experimental results show a major improvement of the stand-alone MPEG-2 Decoder on the TM32 core. The number of cycles needed to decode one coefficient has become even smaller than the number needed for the TM64 core. In addition to this the results show that further optimization of the Entropy Decoder is hardly possible. This can be deducted from the Very Long Instruction Word slot occupancy. This number, over 85%, is exceptionally high and therefore supports the statement that further optimization is hardly possible.

The data management structure in the optimized Entropy Decoder requires a patch to form a bridge between the data management styles. Despite the need of this patch, the performance of the optimized MPEG-2 Decoder in a multi-threaded environment is significant. A gain in performance of 12% on average is measured, where peaks of 20% can be achieved. These results form the lower bound in possible optimization of the MPEG-2 Decoding process. It is required to rewrite the data management structure of the Berkeley MPEG-2 Decoder. This allows a fully performing optimized Entropy Decoder in the MPEG-2 Decoder.

Conclusion

The Entropy Decoding step in the Berkeley MPEG-2 Decoder is optimized for running on processors that are based on parallelism. For this no special instructions in source code are needed. All instructions are written in plain C operations. This makes the optimized source code portable to any platform. Therefore a fair comparison between platforms running the same source code is possible. The Berkeley MPEG-2 decoder is a standard in MPEG-2 decoding. The Entropy Decoding step written for the TM-64 core based on this decoder is improvable. These optimizations are achieved by carefully looking at the function of each step in the decoding process. These steps are analyzed from the rewritten Entropy Decoder based on its TM-64 core version. Important steps to analyze in this case are:

- Steps in which originally super operations are used.
- Steps where 64-bit data types are used.

The super operations are replaced by a choice of three shift operations. The choice between one of the three shift operations is made after all other optimization steps have been implemented. Throughout this implementation results of all three options are compared. At first the If-Else shifting method has the best performance when 64-bit data is retrieved from the lookup table and no pre-loading of data is executing. In the next step the retrieved data from the lookup table is compressed from 64 bits to 32 bits. This has an approximate overall speedup of 1 cycle per coefficient (chapter 6). Here the choice of a shifting method does not make any difference. In the last step, the data treatment changes radically, due to pre-loading of data from the incoming bit-stream. The pre-loading method is possible since it is not on the critical path of the Entropy Decoding process. This critical path is a path of operations where the next operation depends on the previous one. Retrieval of data is such an operation, especially since this data is used in the execution of the next loop. Therefore pre-loading positively influences the critical path. The critical path is shortened since data need not be retrieved from memory but from a register. The data loaded in here is independent of the critical path. The results of pre-loading show a large change in the cycles per coefficient for Entropy Decoding. Grafting results in a new choice of the best shifting method. The little usage of branches in the smart shifting method finally gives its expected result. This is not the case for the question mark shift method. This method is meant to generate guarded operations. These do not appear because of the variance in the guard bit that is needed for this guarded operation.

Two results of pre-loading data from the lookup table with a retrieval of 32 bit data are remarkable.

1. From the 5 available VLIW slots in the TriMedia 32 VLIW core, 4.25 slots get filled on average. This is a slot occupancy of 85% when executing Entropy Decoding on the TM32 VLIW core.
2. Entropy Decoding on the TM32 VLIW core is done in 16 cycles per coefficient on average.

These results are achieved when isolating an Entropy Decoder from the MPEG-2 decoder. When isolating a process from a larger one, the effects of other tasks that are running besides the isolates task are not measured. Since Entropy Decoding is a process that lies on the critical path of MPEG-2 decoding, these effects are minimal. On the other hand, it is shown in the optimization steps that the final version of the optimized Entropy Decoder strongly depends on data management structures. The byte organization in the Berkeley MPEG-2 decoder is completely replaced by an integer organization. This is a main feature of the improvement in performance of the optimized Entropy Decoder.

The Berkeley MPEG-2 Decoder for SpaceCAKE originates from the Berkeley MPEG-2 Decoder. Several instructions that generate multi-threading are added to this decoder. However, the data management is identical to the management in the Berkeley MPEG-2 Decoder. This data organization is managed by separate functions as seen in Figure 7-7 and Figure 7-8. These functions are as dominant as the decoding process. Entropy Decoding shows to be equally important as data management in its number of cycles needed. This problem reduces with the integration of the optimized Entropy Decoder. The optimized MPEG-2 Decoder shows that a lot of cycles can be saved in data management. This results in a boost in performance. The data management structure in the Entropy Decoder is integer organized. This organization forms the basis of its performance. Therefore a patch is needed to form a bridge between the data management styles. Despite the need of this patch, the performance of the optimized MPEG-2 Decoder in a multi-threaded environment is significant. A gain in performance of 12% on average is measured, where peaks of 20% can be achieved. These results form the lower bound in possible optimization of the MPEG-2 Decoding process. It is required to rewrite the data management structure of the Berkeley MPEG-2 Decoder. This allows a fully performing optimized Entropy Decoder in the MPEG-2 Decoder. In conclusion, the optimized MPEG-2 Decoder gives a significant performance improvement based on remarkable results achieved by increasing the efficiency of the compiled Entropy Decoder source code, thus optimizing the Entropy Decoder process.

Bibliography

- [1] *Book 2 - Cookbook Part D, Optimizing TriMedia Applications Software*, July 1999.
- [2] *Book 4 - Software Tools Part A, C Language Users Guide*, July 1999.
- [3] D.Ishii, M.Ikekawa, and I.Kuroda, *Parallel Variable Length Decoding with Inverse Quantization for Software MPEG-2 Decoders*, Proceedings of the IEEE workshop on Signal Processing Systems (SiPS97) (1997), 500–509.
- [4] Basant K. Dwivedi, Jan Hoogerbrugge, Paul Stravers, and M. Balakrishnan, *Exploring Design Space of Parallel Realizations: MPEG-2 decoder case study*, Proceedings of the Ninth International Symposium on Hardware/Software Code design (2001), 92–97.
- [5] E.W.Dijkstra, *A case against GO TO statement*, Commun. ACM **3** (1968), no. 11, 147–148.
- [6] F.Bonomini, F.D.Marco-Zompit, G.Milan, A.Odorico, and D.Palumbo, *Implementing an MPEG-2 Video Decoder Based on the TMS320C80MVP*, Application Report SPRA332 Texas Instruments (1996).
- [7] Robert Haerkens and Joost Sannen, *Variable Length Decoding in software (doc. 176)*, Master's thesis, Fontys Hogescholen, January 1998.
- [8] David A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, In Proceedings of the I.R.E. (1952).
- [9] Yew-San Lee, Bai-Jue Shieh, and Chen-Yi Lee, *A Generalized Prediction Method for Modified Memory-Bases High Throughput VLC Decoder Design*, IEEE Transactions on circuits and systems-II: Analog and Digital Signal Processing **46** (1999), no. 6, 742–754.
- [10] *A quick experiment with a multi-threaded MPEG-2 Decoder for HD video*, <http://sourceforge.philips.com/wwwroot/spacecake>.
- [11] M.Sima, S.Cotofana, S.Vassiliadis, J.T.J. van Eijndhoven, and K.Visser, *MPEG-compliant Entropy Decoding on FPGA-augmented TriMedia/CPU64*, Proc. IEEE Symp. Field-Programmable Custom Computing Machines (2002), 261–270.
- [12] P.Stravers and J.Hoogerbrugge, *Homogeneous Multiprocessing and the future of Silicon Design Paradigms*, Proc. of the 2001 International Symposium on VLSI Technology, Systems and Applications (2001), 184–187.
- [13] S. Rathnam and G. Slavenburg, *An architectural overview of the programmable multimedia processor, TM-1*, Comcon '96. 'Technologies for the Information Superhighway' Digest of Papers (1996), 319–326.

-
- [14] Mihai Sima, Evert-Jan Pol, Jos T.J. van Eijndhoven, Sorin Cotofana, and Stamatis Vassiliadis, *Entropy Decoding on TriMedia/CPU64*, In Proceedings of the Second International Samos Workshop on Systems, Architectures, Modeling and Simulation (2002).
 - [15] *TM1000 Preliminary Data Book*, January 1996.
 - [16] *The PNX1500 Family*, http://www.semiconductors.philips.com/products/nexperia/media_processing/pnx1500/index.html.
 - [17] V.Bhaskaran and K.Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers (1995).

Curriculum Vitae



Arjen Westerterp was born in Stirling, United Kingdom on the 13th of August 1977. In 1989, he started his secondary education at the pre-university school (VWO / A-Levels) 'Stedelijke Scholengemeenschap' in Maastricht where he graduated in 1995. The courses taken for graduation were: Mathematics, Physics, Chemistry, Biology, English, Dutch, French and Geography.

In October 1995, he enlisted as a student at the Faculty of Physics, Edinburgh University. Here he completed his first year of Physics and Electrical Engineering. In September 1996, he enlisted as a student at the Faculty of Electrical Engineering at the Delft University of Technology (T.U. Delft).

In the summer of 2000, he did an internship at KPN Research (now TNO Telecom). The subject of this internship was the exploration of possibilities and boundaries of the Bluetooth network technology. This was analyzed through custom routines developed in the Delphi programming language.

In August 2002, he started his MSc study at the Processor Oriented Architecture group in Eindhoven under supervision of Stamatis Vassiliadis and Jos T.J. van Eindhoven. The Processor Oriented Architecture cluster is part of the Information Technology group. This is a research field in the Information and Software Technology sector from the NatLab of Philips Research Laboratories Eindhoven.